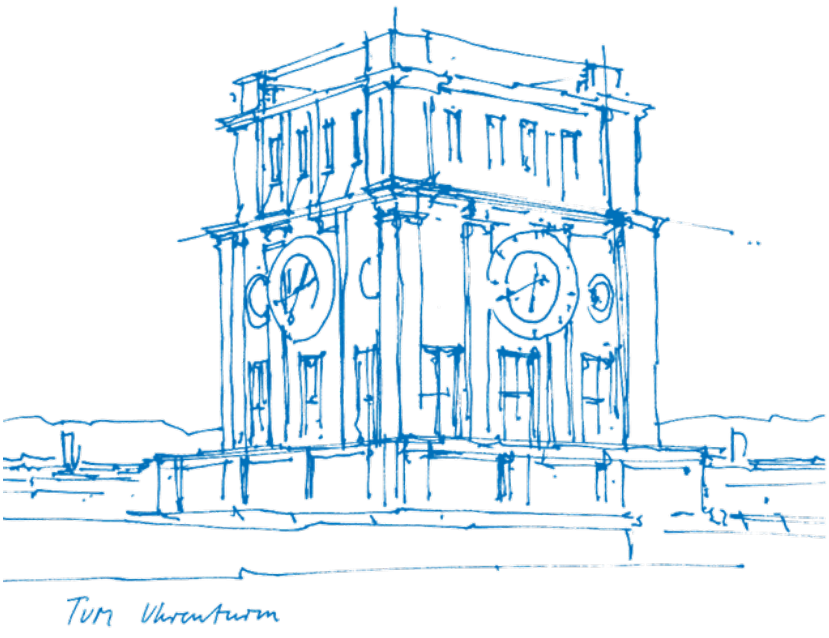
Chair of Information Systems and Business Process Management (i17)
Department of Computer Science
TUM School of Computation, Information and Technology
Technical University of Munich

Master's Thesis in Informatics

Dominik Ingo Voigt

RUST Process Model DSL



Chair of Information Systems and Business Process Management (i17)
Department of Computer Science
TUM School of Computation, Information and Technology
Technical University of Munich

Master's Thesis in Informatics

Dominik Ingo Voigt

RUST Process Model DSL

RUST Prozessmodell-DSL

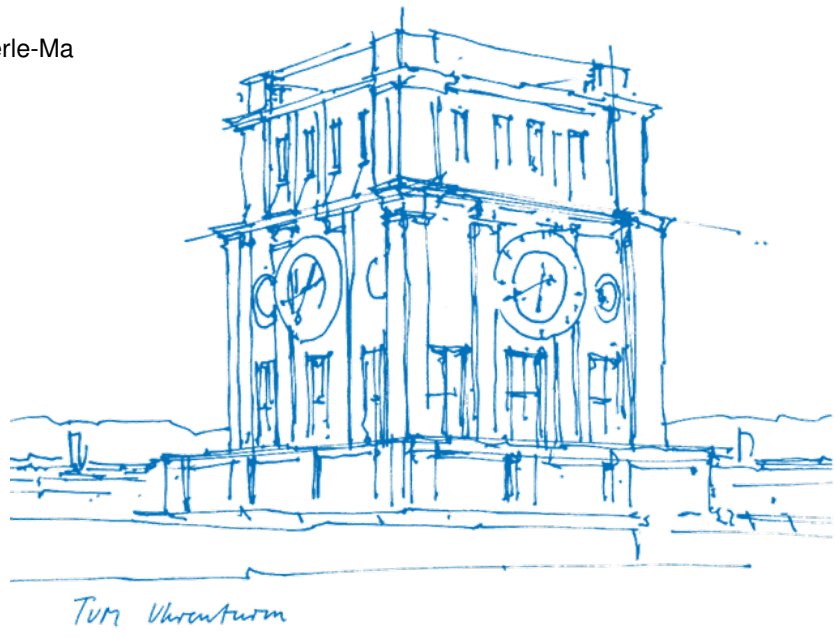
Thesis for the Attainment of the Degree
Master of Science

at the TUM School of Computation, Information and Technology,
Department of Computer Science,
Chair of Information Systems and Business Process Management (i17)

Examiner
Prof. Dr. Stefanie Rinderle-Ma

Supervised by
Dr. Jürgen Mangler

Submitted on
03.02.2025



Acknowledgement

First of all, I want to express my sincere gratitude to Prof. Dr. Stefanie Rinderle-Ma for giving me the opportunity to work on this interesting topic.

Secondly, I would like to thank Dr. Jürgen Mangler for supervising this thesis and providing his continuous support through the whole process. Without "Der Schöpfer" this thesis would not have been possible.

I also express my gratitude towards Lisa and my family who supported me throughout this thesis and kept me, at least partially, sane.

Declaration of Academic Integrity

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

I am aware that the thesis in digital form can be examined for the use of unauthorized aid and in order to determine whether the thesis as a whole or parts incorporated in it may be deemed as plagiarism. For the comparison of my work with existing sources I agree that it shall be entered in a database where it shall also remain after examination, to enable comparison with future theses submitted. Further rights of reproduction and usage, however, are not granted here.

This thesis was not previously presented to another examination board and has not been published.

Garching, 03.02.2025 Dominik Ingo Voigt

Abstract

Business Processes (BP) and Internet of Things (IoT) both have found widespread adoption across industries. IoT-enhanced Business Process Management (BPM) aims to combine these technologies to achieve emergent benefits. The adoption of IoT technology provides many advantages, such as increased data quantity, quality, and recentness. IoT-enhanced BPM leverages these to drive better decision-making processes. However, IoT adoption also creates complex architectural challenges, such as scalability and heterogeneity of device interfaces, as well as operational challenges, such as network connectivity and power supply issues. Business Processes, and thus Business Process Engines, can address these architectural challenges by enabling the orchestration of large-scale service-oriented architectures used in IoT applications.

One promising approach envisioned by IoT-enhanced BPM is modeling the complete business process as a choreography of many smaller business processes for the individual entities involved (e.g., machines and human workers). However, most IoT devices, such as sensor nodes, do not provide enough computing power to host traditional, general-purpose process engines. Thus, this approach requires resource-efficient general-purpose business process engines that enable executing processes on or close to the actual devices. Therefore, this thesis aims to develop a BP engine that allows the execution of business processes on resource-constraint devices, which are common in IoT applications. This solution utilizes a process Domain Specific Language (DSL) and a model-to-code approach to transpile processes into executable, resource-efficient code. This approach is evaluated against a selection of standard business process patterns demonstrating resource-efficient execution and a rich set of supported behavior patterns.

Keywords: *BP Engine, Business Process Engine, IoT, IoT-enhanced BPM*

Contents

List of Tables

List of Figures

1 Introduction

This chapter broadly introduces the research conducted within this thesis. Section 1.1 introduces the research area and motivates the problem's significance. It closes by summarizing the goals and structure of this remaining thesis. Section 1.2 discusses the research questions that this thesis aims to answer. Section 1.3 distinguishes the contributions of this thesis to the corpus of existing research. Section 1.4 introduces the Design Science research methodology and application through this thesis. Lastly, Section 1.6 closes this chapter by outlining the structure of the remaining thesis.

1.1 Motivation

In recent years Internet of Things (IoT) technology has found widespread adoption across many domains, such as manufacturing and healthcare [1], [2]. IoT derives its popularity from the benefits it can provide, such as improved data quantity, quality, and recentness for decision-making and analysis processes [3]. However, IoT integration also introduces many architectural and operational challenges, such as the heterogeneous nature of the different interfaces and capabilities of devices, scalability, unreliable power provisioning, and network connectivity [2], [4], [5].

Business Process Management (BPM) and Business Process Management Systems (BPMS) are nowadays the backbone of many enterprise-scale software [6]. From an architectural point of view, they provide service integration and orchestration, which has become paramount due to the rise of Service-oriented and Microservice architectures in enterprise, specifically IoT-centric applications [7]–[9]. For this, most BPMS leverage a well-defined graphical modeling language such as Business Process Modeling Notation (BPMN) or Business Process Execution Language (BPEL) [6].

Due to their symbiotic capabilities [2], much work has been conducted, both by researchers and practitioners, to combine these technologies [10]. One such research direction is IoT-enhanced BPM, where the integration of IoT sensors and actors enhances a variety of aspects of BPM [2].

Nonetheless, many challenges remain before all the predicted advantages can be realized. For example, in many cases, IoT data and event logs, produced by controlling the interaction of IoT devices through the execution of a process by a Business Process (BP) engine, are still collected

independently (by the devices). Which makes the aggregation, preparation, and use of IoT data for decision-mining and process-mining challenging.

One interesting approach towards achieving the vision of IoT-enhanced BPM is the explicit individual modeling and execution of the underlying process for each process participant, whether it is a human, Cyber-Physical System, or a simple IoT sensor [1], [3]. Closing the gap between IoT and BPM data streams and creating a consistent event stream [2].

However, due to the large number of individual participants, efficient modeling and execution techniques for choreography are required to fully leverage this approach. Furthermore, to execute individual processes in a scalable manner and to address challenges faced by IoT adoption, computing paradigms such as Fog and Mist Computing have to be leveraged [1], [11], [12]. Enabling the execution of Business Processes on resource constrained devices, standard in IoT scenarios [5], requires an efficient and highly stable BP engine [13].

In conclusion, BP engines are often realized as custom interpreters, which makes their use in resource-restricted IoT Fog and Edge environments difficult. There is a real need for a fast and resource-efficient general-purpose BP engine that brings ease of use to the field of IoT.

1.2 Research Questions

This thesis aims to answer three core research questions.

RQ 1 What are the challenges for IoT-enhanced BPM to optimize for CPU and memory utilization?
RQ 2 How can a resource-efficient Process Engine be implemented?
RQ 3 Does the Process engine implemented for **RQ 2** fulfill the identified requirements?

RQ 1 provides an overview of the current research corpus to ensure that the findings and results of the thesis are unique and form a valid scientific contribution. This overview serves as a basis for formulating requirements for the approach chosen in this thesis. For this, a structured literature review was conducted. Section 3 discusses this review.

RQ 2 addresses the need for a stable and resource-efficient process engine identified in prior research. Section 4 discusses conceptual aspects of the proposed engine, such as its general architecture and

qualitative properties. Subsequently, Section 5 discusses implementation details, such as the reasons for selecting the used programming language, frameworks, and technologies.

RQ 3 aims to ensure the scientific rigor of the approach. It ensures that the developed artifact (**RQ 2**) addresses the identified requirements.

1.3 Contribution

This thesis contributed to the field in the following three areas: (1) this thesis identifies existing compute resource-efficient approaches to IoT-enhanced BPM and derives requirements for the to develop a novel process engine, (2) a resource-efficient process engine was developed and is presented, and (3) it provides a detailed evaluation of whether the developed engine satisfies the requirements identified in (1).

1.4 Methodology

As this thesis aims to close the identified gap by developing a new software component, it conducts Design Science (DS) research. Thus, the conducted research follows the Design Science Methodology proposed by Hevner et al. [14]. Subsection 1.4 describes the general DS Framework and the guidelines that DS research, according to Hevner et al., should fulfill. Subsection 1.4 discusses how the research fits into the framework and how it fulfills the proposed guidelines.

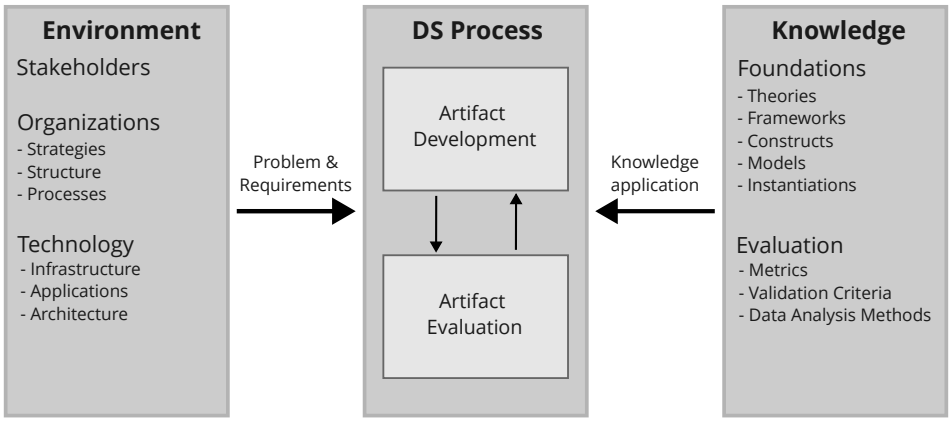
Design Science

Design science is, at its core, a problem-solving paradigm. This research method aims to solve problems through the creation of innovative artifacts. For this, an iterative process of the development and subsequent evaluation of the artifact is performed. An artifact can take different shapes. Most commonly within IS, it is a software artifact; however, it could also be, for example, an evaluation method such as a new metric or analysis method. An artifact is innovative if it solves a yet unknown problem or solves an existing problem better in some aspect than any existing solution [14]. This iterative Development process provides the conducting researchers with a deeper understanding of the problem and the efficacy of their developed artifact.

Figure 1 depicts an adapted representation of the framework for DS research in IS (adapted from a graphic by Hevner et al. [14]). IS research aims to solve problems faced by organizations. The

environment in which problems are identified thus consists of the stakeholders within organizations, the organization strategies, structures and processes, and technological aspects such as the existing Infrastructure, Applications, and the IT Architecture employed by organizations. This environment thus shapes the problems relevant to IS research and further defines the requirements a possible solution has to fulfill. These consist of *quantitative* and *qualitative* properties.

Figure 1
Design Science Framework (adapted from Hevner et al. [14])



Interaction of the Environment and The Existing IS Research with the Currently Conducted Research

Constructs, and *Models* developed by prior research allow to formalize the problem and its possible solution space. Furthermore, employing scientific metrics allows the formalization of artifact requirements. A sufficiently defined problem and solution space allows the development of an appropriate artifact. Leveraging the prior knowledge of the IS research community during the development and evaluation steps ensures the scientific rigor of the approach. Which ensures that the insights gained are reliable and trustworthy. During the design step, this can include aspects such as Constructs (e.g., an Entity in the ER-Model [15]), Models (e.g., the ER-Model [15]), and existing instantiations. Constructs and Models allow for efficient communication of the artifact and allow IS practitioners and researchers to adopt the research artifact. Existing artifacts can be used as a starting point in the solution space to start the search.

After the initial development, to ensure the problem is solved, an evaluation of the artifact is required. Depending on the result, the researchers may return to the design step to improve their artifact. Depending on the problem domain and the artifact developed, an artifact can be evaluated with different methods. The possible table of evaluation methods provided by Hevner et al. [14]

are listed in Table 1. For example, Static and Dynamic Analysis and Simulations can compute quantitative metrics that can be checked against the corresponding requirements.

Table 1
Evaluation Methods by Hevner et al. [14]

1. Observational	Case Study: Study artifact in depth in business environment
	Field Study: Monitor use of artifact in multiple projects
2. Analytical	Static Analysis: Examine structure of artifact for static qualities (e.g., complexity)
	Architecture Analysis: Study fit of artifact into technical IS architecture
	Optimization: Demonstrate inherent optimal properties of artifact or provide optimality bounds on artifact behavior
	Dynamic Analysis: Study artifact in use for dynamic qualities (e.g., performance)
3. Experimental	Controlled Experiment: Study artifact in controlled environment for qualities (e.g., usability)
	Simulation – Execute artifact with artificial data
4. Testing	Functional (Black Box) Testing: Execute artifact interfaces to discover failures and identify defects
	Structural (White Box) Testing: Perform coverage testing of some metric (e.g., execution paths) in the artifact implementation
5. Descriptive	Informed Argument: Use information from the knowledge base (e.g., relevant research) to build a convincing argument for the artifact's utility
	Scenarios: Construct detailed scenarios around the artifact to demonstrate its utility

Lastly, once the evaluation of the artifact is satisfactory, the gained insights must be communicated appropriately. For this, it is essential to note that within IS there are two primary audiences: A technical audience, referred to as IS practitioners, and a managerial audience. Since both audiences have different goals and backgrounds it is important to communicate the findings appropriately.

For IS practitioners and technical researchers, the problem context and solution must be communicated with enough detail to allow for re-implementation and integration into the target environments. This report must document the search process, the artifact's construction, and the evaluation process.

For a managerial audience consisting of stakeholders with decision-making capabilities within companies and management-focused researchers, the solution must be communicated with an emphasis on the importance of the problem and the value of a solution. Nonetheless, the artifact must be described in enough detail to ensure that the audience can understand its application and function within the application context.

Application

This thesis tries to address the challenges of IoT-enhanced BPM. In particular, this thesis attempts to address the demand for a general-purpose resource-efficient process engine that can be deployed on IoT devices. The novelty of this artifact is described in Section 1.3.

The stakeholders for such an artifact are Information Science (IS) practitioners (Process designers and supervisors) at organizations that aim to leverage the advantages of IoT-enhanced BPM and want to deploy Business Processes directly on IoT devices. This deployment of Business Processes on resource constrained devices results in the following requirements. The Process Engine must be able to execute fully featured BP models. For the definition of fully featured, this thesis uses the features supported by the CPEE as defined by Mangler et al.[16], [17]. It needs to be more resource-efficient than comparable existing solutions. Lastly, it needs to be stable, easily modifiable, and extensible. A more exhaustive set of requirements is introduced in 4.1.

For the Business Models, we leverage the existing BP Notation used by the Cloud Process Execution Engine (CPEE)[1] and use the CPEE architecture and code as a basis for our implementation. As conformance to the interfaces used within the CPEE allows direct support for modeling and supervision tools available for the CPEE[2].

Section 1.5 discusses the evaluation of this artifact.

1.5 Evaluation

For the evaluation, the requirements described in the Application Section above need to be checked. For the BP pattern support, this becomes as simple as checking. The fulfillment of the qualitative requirements by the artifact will be discussed in Section 4. For the resource efficiency, this becomes harder. While there are approaches at resource-efficient process engines, such as [18]–[21], comparing quantitatively against these is difficult. The engines provide different features, execution strategies (compiled vs. interpreted), and programming languages. Moreover, most papers provide no measurements of resource utilization for their approaches. As such, to compare en-

[1] `https://cpee.org/`, last access: 2025-01-01
[2] `https://cpee.org/flow/`, last access: 2025-01-01

gines with equivalent feature sets, we decide to compare our solution against the CPEE reference implementation[3]. For the other engines, the available data will be compared where possible.

For the direct resource efficiency comparison, a set of process patterns (see Section 6.1 for more details) are modeled using the CPEE notation and executed on the reference and proposed implementations. Both implementations are evaluated on a desktop computer with an AMD Ryzen 7 8745H and 32 GB of RAM (furthermore called ECHO) representing an unconstrained computing platform, and an Orange Pi Zero 2W[4] representing a resource constrained platform. Memory and execution time are measured for all patterns.

1.6 Structure

The remainder of this thesis is structured as follows. Chapter 2 provides introductory background material, which will be the basis for the remaining thesis. It will introduce and define topics relevant to this thesis, such as IoT, computing paradigms, and BPM. Chapter 3 introduces and discusses related scientific approaches and identifies the research gap this thesis aims to address. Chapter 4 discusses the solution conceptually. This includes the architecture of the software artifact as well as the features it supports. Chapter 5 discusses implementation details, such as the programming language employed and concurrent interactions based on an example. Chapter 6 evaluates the developed artifact in light of the requirements identified in Chapter 3. Chapter 7 provides a summarizing overview of this thesis, including where the different research questions are answered and where the identified requirements are evaluated. Lastly, Chapter 8 summarizes the findings of this thesis and provides areas for future work.

[3] `https://cpee.org/`, last access: 2025-01-01

[4] `http://www.orangepi.org/html/hardWare/computerAndMicrocontrollers/details/Orange-Pi-Zero-2W.html`, last access: 2025-01-01

2 Background

This section introduces different topics relevant to this thesis. Section 2.1 and Section 2.2 provide a short definition of Business Processes and IoT technology. Section 2.3 will introduce different Distributed Computing Paradigms. Lastly, Section 2.4 provides a short definition of a Domain Specific Language in the context of Computer Science and Information Systems.

2.1 Business Process Technology

A Business Process is a sequenced collection of activities, decisions, and other control flow constructs and events [22], [23]. Business Process Management is the collective term for the activities conducted during the different lifecycles of a process, such as the modeling, execution, adaption, and monitoring [6].

Business Process Management Systems (BPMS) are Information systems that support BPM on an operational level [3], [13]. They often provide graphical user interfaces for the modeling of Business Processes, an engine for executing Business Processes, and some facility to monitor running instances.

Business Process Modeling Languages, such as BPMN [24] and BPEL [25] are graphical Domain Specific Languages (DSLs) that provide well-defined sets of components to model Business Processes. The advantage of such graphical modeling languages is their abstraction from low-level programming languages [1]. Which leads to easier maintainability and adaptability of existing processes. Due to this, BPM has been widely adopted across many domains [26].

These Graphical Models are then executed by Business Process Engines, which can either (1) translate the graphical process model into executable code, or (2) execute the model by interpreting it. Both options have certain advantages and disadvantages, which Chapter 4 further discusses.

2.2 Internet of Things (IoT)

In its original inclination, IoT referred to leveraging real-world data through computing devices without human intervention to reduce cost and increase efficiency [27]. Nowadays, the meaning of

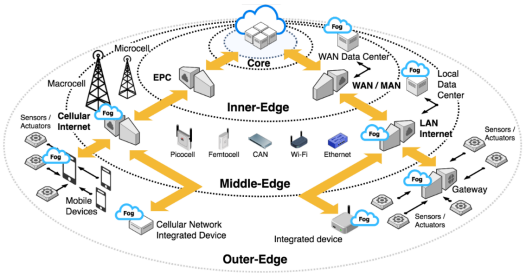
IoT has expanded. Modern IoT applications leverage various of physical devices with integrated computing and networking capabilities [5]. These devices have different granularities and can range from individual SOC devices to complex Cyber-Physical Systems with an exposed network interface. Generally, IoT devices fall into two categories. Sensors (e.g., temperature, vibration, imaging, humidity) allow to sense the environment and actuators (e.g., motors, fans, heating elements) [28] allow to influence it. Notably, certain devices with more advanced capabilities, such as smart watches, can act as sensors (e.g., heart rate) and actors (e.g., providing visual or auditory cues). Utilizing the rich set of featured provided by IoT devices allows for a wide range of application scenarios, such as real-time monitoring and remote control [29].

Due to the wide range of capabilities and, thus, application scenarios, IoT has become a core technology across domains [1]. While IoT devices are often inexpensive, they only provide limited computing capabilities [5]. To counteract this, many IoT applications initially leveraged Cloud Computing offerings to handle compute-intensive workloads. However, due to the increasing need for scalability due to rapid adoption and other challenges such as latency and privacy [5], [11] other distributed computing paradigms such as Fog and Mist computing are leveraged as they provide answers to these challenges [8], [12].

Challenges

While adopting IoT technology provides significant possibilities due to the increase in capabilities to sense and influence the environment, it also introduces challenges of which a selection from existing research is shortly introduced. Due to the large quantity of sensors and actuators used within IoT applications, significant amounts of data are produced [2], [4], [5], [11]. Transmission and analysis of this data in a scalable fashion, especially under latency constraints [5], is challenging [2] (**C1**). The heterogeneous nature of sensors introduce further challenges. Events emitted by sensors can have different granularities and might, on their own, not provide any actionable information, requiring the need for refinement and analysis [2], [3] (**C2**). While modern IoT devices offer increasing amounts of computing capabilities the overall quantity of computing resources on these platforms is still constrained compared to desktop environments, making utilization of these computing resources more challenging (e.g., due to the general lack of an Operating System or the restricted feature set of the provided OS) [5] (**C3**). Within many deployment scenarios, IoT devices have an unreliable

Figure 2
The Computing Hierarchy as defined by Chang et al. [5]



power supply (e.g., due to being battery-powered) (**C4**) and unreliable network connections (**C5**) [4], [5].

2.3 Computing Paradigms

Computing Hierarchy

Chang et al. [5] define a computing hierarchy with the cloud at its core. Figure 2 shows this hierarchy. The different edge tiers describe conceptual layers between the cloud and the device the application is running on. Notable here is that each of these tiers has different properties regarding cost, latency, and computing capabilities. While the cloud and WAN Data Centers have more computing capabilities than almost all local data centers and are cheaper due to economy of scale effects, they lose regarding latency and scalability considerations. As such, to build scalable and cost-efficient IoT applications utilizing all possible computing resources across different tiers become paramount.

Cloud Computing

Cloud Computing generally refers to the on-demand provisioning of virtualized computing resources within globally distributed data centers by cloud computing providers to customers, commonly on a pay-per-use basis [30]. Cloud providers offer an ever-increasing range of services that can be categorized into the following general categories. Infrastructure as a Service (IaaS), Platform as a Service (PaaS) and Software as a Service (SaaS). These categories all provide IT services in increasing abstraction. Infrastructure as a Service provides direct access to computing and storage resources. PaaS provides already configured Execution Platforms that allow rapid development and deployment of applications. Examples of this are Databases (e.g., AWS Aurora) and Middleware

offerings (e.g., Azure Event Grid). SaaS provides the hosting of Software applications. Examples of this are Google Docs and Sheets.

While cloud computing provides possibilities for rapid and on-demand deployment of computing resources and IT services, there are also specific issues. To ensure scalability of IoT architectures, not all data can be transferred to cloud offerings for processing. Such an approach could rapidly lead to unmanageable service loads due to the high volume of sensor readings. A prime example are video-based applications that provide large amounts of data and are expensive to process [31]. Due to this, a localized approach is required. Furthermore, due to the centralized nature of cloud offerings, latency requirements of certain applications become unsatisfiable using cloud computing offerings due to the large number of network hops and network unreliability.

Edge Computing

Edge computing refers to the provisioning of computing and storage nodes at the edge of the internet close to consumers and producers (e.g., sensors or smart devices) [31]. These computing nodes are referred to as Fog nodes or cloudlets. Thus, computing resources would need to be located within the Inner-edge of the computing hierarchy (ref. Fig. 2). However, these resources are often located within the LAN network co-located with the other devices, i.e., within the Middle-edge.

This proximity of computing and storage resources to the producers and consumers has several advantages. It reduces the latency significantly due to the drastically reduced number of hops [31], [32]. It can address privacy and other (regulatory) concerns due to the necessary geographic proximity [31], [32]. Lastly, the scalability of IoT applications is significantly improved as data is analyzed and filtered close to the source and only the relevant information is transmitted to the cloud [31].

Fog and Mist Computing

Fog computing can be seen as a generalization of edge computing as it describes a design where computing devices along all tiers of the hierarchy are leveraged to achieve latency, privacy, and scalability requirements of modern IoT applications [5], [32]. In a Fog Computing oriented IoT architecture devices in the Outer-Edge are used to sense and influence the environment (IoT Sensors and actuators) and handle computationally inexpensive tasks on Gateway devices (e.g., Protocol

bridging Devices). Tasks that are computationally expensive, such as data filtering and analysis tasks (e.g., video streams [33]) that cannot be passed onto the cloud due to scalability, privacy, or latency requirements are handled by computing devices/services within the Middle- or Inner-Edge tiers[4], [32].

Mist Computing has emerged due to the increasing computing capabilities of IoT devices (e.g., more powerful micro-controllers) [5], [30]. Here, the device handles specific computing tasks. This further increases scalability as less data is transferred over the network to the Middle- or Inner-Edge tiers [4], [11].

2.4 Domain Specific Language (DSL)

A Domain Specific Language (DSL) provides a syntactically and semantically well-defined language tailored to a specific domain [34]. While these languages are more constrained than general-purpose programming languages, they are closely aligned with the problem at hand [35]. Thus increasing the speed and efficiency of developing a solution and maintainability due to providing simpler solution formulations [36]. Examples are BPMN 2.0 for describing business processes or Verilog for hardware design.

DSLs can be implemented as Internal or External DSLs [35]. External DSLs are self-sufficient and provide their own parser. On the other hand, Internal DSLs are defined within another language (most commonly general-purpose programming languages) and utilize the existing infrastructure and features provided by the host language [35]. This provides an idiomatic way of formulating solutions for the specific domain of DSL. An example of this is the Markup UnderScore DSL [5] is written in the Ruby[6] programming language. It contains a DSL to generate HTML and JSON documents in an easy-to-read way.

[5] `https://github.com/etm/markus.rb`, last access: 2025-01-01
[6] `https://www.ruby-lang.org`, last access: 2025-01-01

3 Related Work

This section discusses the literature review conducted to answer **RQ 1**. Section 3.1 describes how each step of the literature review was conducted. Section 3.2 discusses the directly related work identified.

3.1 Methodology

This literature review was conducted with the help of JabRef[7], performing the following steps. Firstly, search queries were defined based on a set of papers found through ad hoc searches and consulting an experienced researcher in the domain. Secondly, these queries were used to search for papers on Google Scholar[8]. While initially only title-only queries were considered, three full body queries were added to improve the chances of identifying relevant papers. The top 50 hits based on Google Scholars ranking were considered for these search queries. Subsequently, each paper was filtered based on these criteria: Title, Abstract, and Conclusion. Thirdly, backward snowballing and filtering were conducted. Lastly, all relevant papers were identified. Here, a distinction was made between general approaches for IoT-enhanced BPM and specifically resource-efficient approaches. While general approaches were collected to get an overview of IoT-enhanced BPM approaches, these papers will not be discussed. The discussion in Section 3.2 will focus on resource-efficient approaches. Here, every identified paper will be individually discussed. A summary of the search and selection results is provided in Table 2. The query identifiers in the table correlate with the identifiers used in the following subsections.

Title-Only Queries

(Q1) Mobile ("Business Process Management" OR BPM)

(Q2) Mobile Process Engine

(Q3) Transpilation

(Q4) Edge Process Management

(Q5) (BPM OR "Business Process") Engine

(Q6) Workflow Engine (Process OR Business)

[7] `https://jabref.org`, last access: 2025-01-01
[8] `scholar.google.com`, last access: 2025-01-01

Table 2
Search & Filtering Results

Query Identifier	Number Hits	Relevant
Q1	26	1
Q2	4	3
Q3	40	0
Q4	15	2
Q5	38	0
Q6	21	2
Q7	9	0
Q8	7	3
Q9	4	2
Q10	> 50	3
Q11	>50	19
Q12	35	3
Snowballed	-	22

(Q7) Smart Manufacturing ("Business Process" OR BPM OR Workflow)

(Q8) (IoT OR "Internet of Things") ("Business Process" OR BPM OR Workflow) engine

(Q9) (IoT OR "Internet of Things") ("Business Process" OR "BPM") (SLR OR "Systematic Literature")

Full-body Queries (TOP 50)

(Q10) ("Business process" OR BPM) DSL (Map OR Translation OR Transpilation OR Executable)

(Q11) (IoT OR "internet of things") ("business Process" OR BPM)

(Q12) (Fog OR Edge OR Cloud OR Mist OR SOA OR "Service Oriented") (Computing OR Architecture) (BPM OR "Business Process" OR Workflow) engine

3.2 Analysis

During the search, 11 scientific works were identified that deployed process engines on mobile or embedded devices and thus had restrictive resource-restrictive execution environments. First, each of the papers will be summarized. Then key aspects of these papers will be discussed. Where available, the Memory requirements will be highlighted and discussed. A summary of the key characteristics is provided in Table 3. Lastly, the novelty of the solution artifact presented in Sections 4 and 5 will be discussed.

Table 3
Summary of related Mobile/IoT Process Engines

Name	Progr. Language	Supported Process Model	Connectivity	Mem.	Comp. or Interpr.	Execution
Sliver [19]	Java	BPEL	SOAP	<22 MB	Interpreted	Centralized
Pajunen et al. [18]	Java [1]	BPEL	SOAP	?	Interpreted	Centralized
CiAN [37]	Java	BPEL	SOAP	?	Interpreted	Distributed
EMWF [20]	C	augmented SISARL-XPDL	-	1MB	Compiled	Centralized
ERWF [38]	C	augmented SISARL-XPDL	-	8 MB	Compiled	Centralized
MARPLE [39]	.Net	ADEPT	Standalone	?	Interpreted	Distributed
Glombitza et al. [40]	C++	BPEL	SOAP over LTP	1 KB (???)	Compiled	Centralized
SPiCa [41]	Objective-C	BPEL	HTTP	?	Interpreted	Distributed
SCOPII [42]	?	BPEL	HTTP	?	Interpreted	Distributed
AMSNP [43]	Objective-C	BPMN	HTTP	?	Interpreted	Distributed
Schobel et al. [21]	Java	ADEPT2	?	?	Interpreted	Centralized

Hackmann et al. [19] implemented a Java-based BPEL workflow engine for mobile devices called Sliver. Their work focused on efficiently implementing such an Engine with SOAP support. This is achieved by using the lightweight kXML and kSOAP libraries. Contrary to their work, the engine developed for this thesis provides support for HTTP-based communication as it is the de-facto standard for IoT applications [44] and follows a transpilation approach that compiles the BP model into an internal DSL that is then compiled. This approach generally achieves better resource efficiency as direct compilation allows for optimization for the processor architecture of the target device and does not require an additional software layer such as a virtual machine or an interpreter [45]. While they provide no information on the memory consumption of their approach, they state that their approach outperforms on the memory consumption of ActiveBPEL of 22 MB.

Pajunen et al. [18] follow a similar approach to Hackmann et al.. However, they focus less on the resource-efficient implementation of the engine and more on support for local services and different communication protocols such as HTTP, SMS, and email. They appear to also use Java and follow an interpreted approach. As such, the developed artifact differs from this work in the same way.

Sen et al. leverage the Sliver engine for choreographic workflow execution in Mobile Ad-hoc Networks (MANETs) called CiAN. In their paper, multiple nodes each host their own engine instance (Sliver) and execute a choreographed workflow together in a peer-to-peer manner. While such an extension of workflow engines has value for IoT applications as MANETs are commonly used in such an environment. However, for individual nodes, the same differences arise due to the use of an interpreted approach and reliance on a virtual machine.

Chou et al. [20] developed a C-based embedded workflow engine for robotic applications supporting an augmented subset of XPDL called EMWF. Very similar to our approach, models would be compiled into machine code. However, this engine only provides support for local activities and does not provide any form of connectivity. While executing a workflow, the engine consumes about 1 MB of memory. A very similar approach was followed by Chen et al. [38] on their embedded process engine called ERWF. Their work additionally provided real-time execution support. Their engine consumed about 8 MB of memory while running a workflow. While both of these works also follow a model-to-code approach and are highly memory efficient, they cannot be used as a general-purpose process engine due to the specific feature set supported and the missing network

connectivity. Furthermore, it does not provide an intermediary internal DSL for the Model to code transpilation.

Pryss et al. [39] proposed a distributed workflow engine for a seemingly custom process definition language using .Net called MARPLE. The MARPLE architecture allows workflows to be executed as workflow fragments in a choreography by individual nodes run their own engine. Deployment, relocation of workflow fragments, and execution of web-service calls are handled by a central "Mediation Center" component. Communication is facilitated through a stand-alone protocol [1]. Process models are executed through interpretation this is contrary to our approach. Furthermore, due to the use of the .net framework, execution environments require the Common Language runtime. Pryss et al. provides no resource utilization measures for their approach.

Glombitza et al. present a model-to-code approach for the execution of BPEL models on Sensor Nodes. In their approach code is generated from the BPEL model for the target platform. Their approach replaces HTTP communication with LTP. Due to this, realizing impressive memory efficiency at the cost of compatibility or additional gateway services as most of the Web utilizes HTTP-based communication. Additionally, their approach does not include an intermediary internal DSL that simplifies the maintainability of the generated code.

Chang et al. [41] developed an application framework for the Social Private Cloud called SPiCa. In their approach, individual devices host a SPiCa instance, including a BPEL workflow component. Each node also has a mediator component that allows the workflow component to dynamically offload computationally intensive tasks to other SPiCa nodes (directly connected local devices) or the Public Cloud. A prototype was developed for iOS devices such as the iPhone 4S. Chang et al. provided no resource utilization measures for their approach. Conceptually similar works have been proposed by Chang et al in [42], [43]. In all three approaches, resource-efficient process execution was not actively discussed and no resource utilization measurements were reported.

Schobel et al. [21] developed a lightweight process engine to execute so-called instruments, which can be used for medical surveys (including sensor measurements) in Java. These instruments are modeled within a graphical modeling language and then transpiled into a process model (ADEPT2) that is then executed on mobile devices. The engine supports offline execution of processes. It also provides execution of subprograms embedded in the instrument called executable components

for execution of domain-specific features, e.g., sensor measurements. The utilization of Java leads to the same resource efficiency concerns as with Hackmann et al.'s approach. Furthermore, the seeming lack of connectivity of the workflow engine does not allow for the use of the engine for general-purpose process execution. Schobel et al. provided no resource utilization measurements.

3.3 Distinction from existing Research

While prior works focused on specific details of mobile process execution, e.g., Schobel et al. [21] supporting offline execution, Glombitza et al. [40] and Chou et al. [20] focus on resource-efficient execution using a process-to-code approach, no other work to our knowledge combined approaches in the way of the proposed solution. The three properties of the solution presented in this thesis are: (1) A model-to-code approach that translates the process model to executable code, and specifically an internal DSL as in [16], [17] that can be compiled and natively executed on devices with low compute capabilities. (2) Provide full HTTP support to communicate with external web services, including HTTP Multipart. (3) Full logging of the process execution [23] via a distributed engine architecture. Full logging of the process execution allows for monitoring, stopping, modifying (control flow, execution context, and even the process model), and resuming the process [16] to recover from errors during process execution. Furthermore, this allows recovery after system failures, e.g., due to power outages common in IoT scenarios [4].

The solution presented in this thesis retains all the properties of the implementation by Mangler and Stürmer [16], [17], while improving the resource utilization by a factor of up to 13.

4 Solution Design

This chapter discusses the proposed solution in detail. Section 4.1 documents the quantitative and qualitative requirements the proposed solution artifact should satisfy. Section 4.2 discusses the general architecture upon which the workflow engine is built. Including the interfaces with external components. Section 4.3 discusses the general DSL language components that are required to support the description of general-purpose business processes.

4.1 Requirements

This thesis aims to develop of an engine that can execute general-purpose Business Processes, specifically IoT-related processes, on resource-constrained devices. The requirements are derived from the related work on process engines on resource constrained devices (see Section 3.2 and from IoT application-specific Challenges.

To enable the execution of business processes on IoT devices, resources must be utilized efficiently to address the limited compute resources available (see **C3** in Section 2.2) (**R1**). For this memory consumption and process execution time are used as quantitative measurements. Memory consumption for individual instances should be below 10 MiB. While execution time should be significantly faster than an existing reference implementation, that is not focused on resource-efficiency. To enable monitoring logging of the process execution by the DSL implementation is essential (**R2**). This further allows the engine to stop and resume processes and freely modify the process model, the execution context, and the thread of execution [16]. This further allows, while not realized in this implementation, instances hosted on IoT devices to resume process execution after an unexpected crashes e.g.m due to a power outage (see **C4** in Section 2.2). To ensure the efficient execution of process models a transpilation-based approach as used by Stürmer et al. [16] and Glombitza et al. [40]. To ensure the maintainability of the produced instance code artifacts [35], an internal DSL within a host programming language is used, as proposed by Stürmer et al. [16]. The DSL, thus, should be easy to understand and extend and be powerful enough to express common workflow patterns (**R3**). Lastly, to ensure that the process engine can be used to support a wide variety of application scenarios (general-purpose), HTTP as the de-facto standard protocol in the Internet should be supported including HTTP multipart support (**R4**).

4.2 Architecture

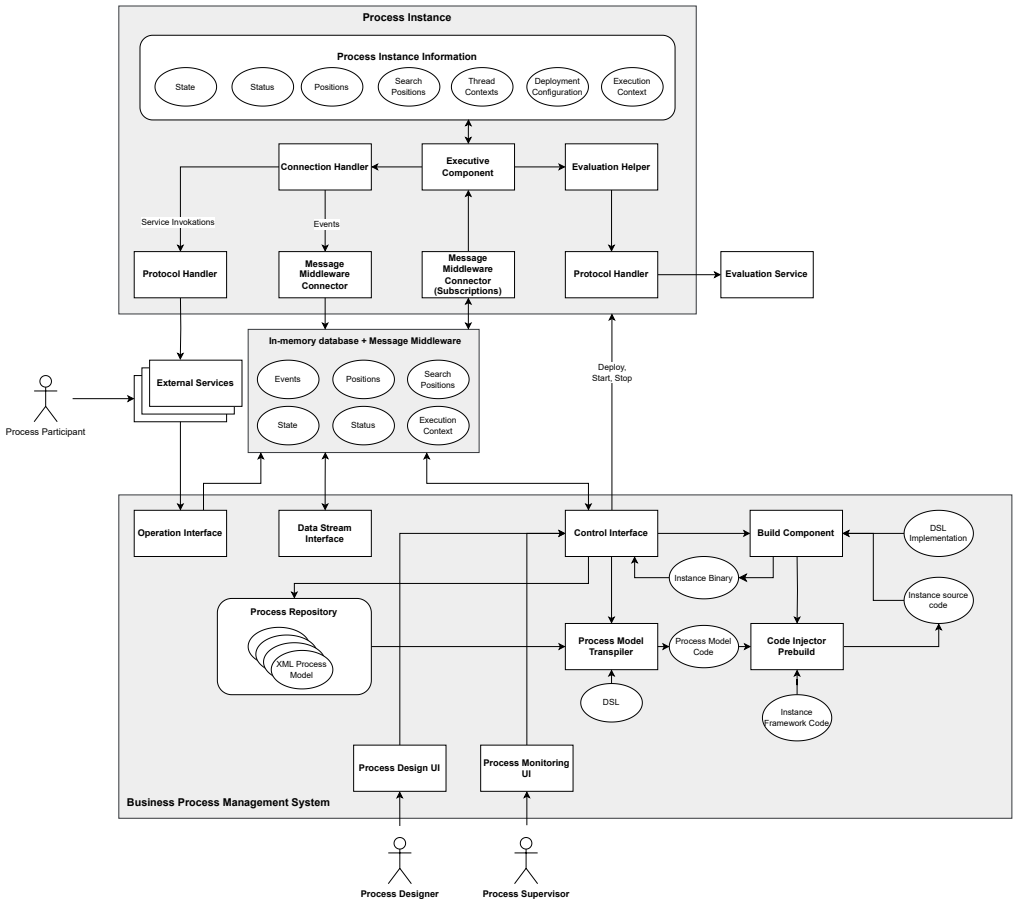
The architecture of the process instance follows the architecture described by Stürmer et al. [16]. The interfacing of the process instance that executes the process with the BPMS component that handles logging, monitoring, and asynchronous callbacks follows the architecture described by Mangler [23]. The reason for this is that this allows the process instance, which can be run on any edge tier within the computing hierarchy, or even on the device, to directly interact with the existing BPMS component that is hosted on an arbitrary server (inner- or middle-edge). This provides modeling, deployment, and monitoring features provided by the BPMS component [23]. Furthermore, the BPMS developed by Mangler [23] supports the model-to-code approach just by providing a transpilation unit fulfilling the model-to-code requirement (**R1**). The architecture for this is depicted in Figure 3. This architecture is discussed in tehf irst subsection by providing a walk-through based on how a business process would be modeled and subsequently transpiled, deployed, and executed. The second subsection discusses architectural decisions and emergent properties. The last subsection discusses the criteria considered when selecting a host language.

Architecture Walk-through

A **Process Designer** can utilize the **Process Design UI** to model the business process graphically and enrich the model with execution details such as service endpoints, data (stored as named variables), together referred to as the execution context, and handlers for different scenarios. The result of this step is an *XML Process Model*. Which is stored within the **Process Repository**.

Subsequently, a **Process Supervisor** can execute the business process via the **Process Monitoring UI**. Which invokes the **Control Interface** which in turn passes the process model to the **Process Model Transpiler** that transpiles the *XML Process Model* to an equivalent program in the internal DSL is described in Sections 4.3 and 5.2. Thereafter, the **Build Component** is invoked by the **Control Interface** to create the executable process instance. For this, the *Process Model Code* is injected into the *Instance Framework Code*. This artifact is then compiled with the *DSL implementation* for the target architecture at hand. The resulting *Instance binary* contains the logic process model and all the operative code like logging and protocol handlers and can thus self-sufficiently execute the process model. Lastly, the binary, execution context, and configuration files are deployed on the target, and the binary is executed.

Figure 3
Architecture of the proposed Artifact and Interaction with the existing BPMS



Process execution on the target starts by loading the configuration. This configuration is split into the *Execution Context*, which includes the execution details, such as endpoints and data provided to the process, and the *Deployment configuration* containing information, such as the location of the **Message Middleware and In-Memory-Database** and the unique instance identifier. Thereafter, the **Executive Component** is initialized. This component semantically represents the process as a whole. All *Process Instance Information* is managed by this component, and the DSL is defined as methods on the **Executive Component** instance. Subsequently, a **Message Middleware Connector** is initiated to listen for events emitted by the management component. This is used for features such as callbacks for asynchronous activities [23]. After this execution of the process model DSL code begins.

For each Activity a **Connection Handler** is initialized. This includes the invocation of external services (see the Call DSL component in Section 4.3) as well as the execution of script artifacts (see Manipulate DSL Component in Section 4.3). The Connection Handler encapsulates all communication between the instance and external components, including external services and the middleware component. The tasks of the Connection Handler consist of the emission of events, e.g., position changes or changes in the Execution Context through Handlers, for logging (**R2**) and the creation and use of a **Protocol Handler** instance to invoke external services such as creating tasks in a work list for Process Participants or invoking other functionalities. It is important to note here that external services handle most process-level functionality such as timeouts and subprocess creation. This architectural decision keeps the engine lightweight and easily extensible [23].

If any handlers, realized as code artifacts, need to be executed, e.g., to prepare data for service invocations, they are passed to the **Evaluation Helper**, which handles the invocation of the corresponding **Evaluation Service** via a **Protocol Handler** Instance. The Evaluation Helper subsequently handles the returned result. Such results can include the manipulation of endpoints or data elements via the handler code or signaling to retry execution or to stop the process instance execution. Section 5.3 discusses Signals further.

Architectural Decisions and Properties

The core motivation behind the architecture chosen was to ensure easy extensibility and resource-efficient process execution with minimal requirements while providing the features of a general-

purpose process engine. Each of the following sections discusses a specific architectural decision and the resulting properties.

The communication between the BPMS and an instance is facilitated completely via the Message Middleware and In-Memory-Database component (excluding deployment, start, and stop of each instance).

This decision also decouples the **Process Instance** from the **BPMS** and thus allows the deployment of the individual components on possibly different tiers of the computing hierarchy (see Section 2.3). This allows deploying instances directly on the IoT devices, while hosting the **Middleware** and the **BPMS** on the middle- or inner-edge tiers. Thus enabling Mist Computing architectures for IoT-enhanced BPM applications. Additionally this decision allows for horizontal scalability of certain services of the BPMS, such as the logging service by decoupling the event source (instance) and sink (BPMS).

Furthermore utilizing an in-memory database as a middleware component, combined with an appropriate DSL implementation, allows the complete application state of every instance and the corresponding execution logs to be automatically persisted during execution, satisfying the logging requirement (**R2**) (ref. Section 4.1). This feature, combined with the existing **BPMS** implementation by Mangler[23], realizes useful features. Firstly, it allows the instance to recover in case of unexpected crashes, e.g., due to power outages common in IoT environments [4], and resume process execution at the last logged position (see **C4** in Section 2.2). Secondly, it allows stopping instances during execution and adapting the control flow, execution context, and even the process model to handle errors occurring during process execution, e.g. due to divergence of the process model from reality [3].

The execution of handler code via the Evaluation Helper by invocation of the Evaluation Service has certain advantages and disadvantages. While this increases latency and the potential for network-related issues, it provides significant advantages. Firstly, this allows the formulation of handler code in arbitrary programming languages that do not have to be supported on the deployment target. Simplifying development of Processes by Process Designers. Secondly, the possibility for handler code to rely on arbitrary execution environment requirements (e.g., access to a GPU). Lastly, the possibility to either link the code directly into the *Instance Binary* or to deploy the execution service

on the IoT device by replacing the Protocol Handler component for scenarios where latency and network concerns are predominant.

Language Selection Criteria

Selecting an appropriate DSL host programming language is vital to ensure that the resulting instance code satisfies the requirements identified in Section 4.1. The first criterion is the resource efficiency of the language (**C1**). The second criterion is the capability of the language to define a DSL that is easy to understand, and the resulting code fragment is easy to modify (**C2**). The third criterion is ease of deployment, whether the language requires an additional runtime (interpreted) or whether the build system allows for dynamic and static linking (compiled) (**C3**). The final criterion is how easy it is to write memory-safe code to ensure stability and maturity of the implementation (**C4**).

4.3 DSL

To ensure that the engine supports the modeling and execution of general-purpose business processes, the DSL has to be defined accordingly. Thus, every component of a business process's control flow needs a corresponding DSL element. Since our solution utilizes the CPEE for modeling, the DSL is designed to support the modeling constructs it supports and thus is based on the existing DSL for the CPEE as proposed by Mangler et al. [17]. The advantage of this is that Mangler et al. already evaluated the DSL's expressiveness, demonstrating a general-purpose level of expressiveness as it supports all Basic Control Flow Patterns and a wide range of more advanced patterns [17]. Thus satisfying requirement **R3** by design. Note that the specification of DSL is deferred until Section 5.2.

Firstly, the DSL components proposed by Mangler et al. will be discussed. An Activity (Task in BPMN) represents a single step within a Business Process. Activities can either call external services that implement the functionality, represented by the **call** component in the DSL, or implement the functionality via a code artifact, represented by the **manipulate** component. These are the components that delegate further to the **Connection Wrapper Component**.

Parallel Gateways allow the parallel execution of multiple branches simultaneously. A parallel gateway with its multiple branches is represented by the **parallel_do** component with an equivalent amount of **parallel_branch** instances nested within. The parallel gateway can further be assigned

a wait threshold and wait type. If neither is specified, all **parallel_branch**es will execute until they terminate. If the wait threshold is specified (as a positive natural number), only the specified number of branches are guaranteed to execute until termination. These branches are the ones that, depending on the wait type, either finish their first tasks or their last task first. All remaining branches still executing when the wait threshold is reached will finish their currently executing task and terminate. All branches are realized, provided the host language provides the concept, as threads.

Decision Gateways allow the selective execution of branches based on a per-branch defined condition. A Decision Gateway with multiple alternative branches, each annotated with a condition and an optional else branch is represented by the **choose** component with nested **alternative** components and an optional otherwise component. Decision Gateways offer two modes of execution to model the Exclusive and Inclusive Gateway defined in BPMN 2.0 respectively referred to as the exclusive and inclusive.

Loops allow the repeated execution of the contained process fragment. The **loop_exec** component represents this concept. Loops can either be top-controlled (while) or bottom-controlled (do-while). Top-controlled loops test the condition before every execution of the body. A bottom-controlled loop tests the condition after every execution and thus executes the body at least once. Both types of conditions are represented by **pre_test** and **post_test** components respectively.

A critical section allows to mark process segments within a parallel gateway. A critical section uses a Mutex to guarantee that only a single thread at a time can enter a marked process segment. While a thread executes a critical section, all other threads that want to enter a process segment of the same critical region are postponed. Such a critical section protecting a process segment is represented by the **critical_do** component with the process segment's DSL code nested within. The Mutex implementation relies on the synchronization options structures provided by the host programming language.

Since Mangler et al. [17] proposed this initial DSL, the CPEE was extended by further modeling concepts. Stop and Terminate allows to stop or terminate the process model execution by the process model itself. This could be useful for cases where, in the case of stop, human intervention is required due to an unpredicted error. These are represented by the **stop** and **terminate** components. Escape

allows to terminate the execution of the body of the loop and to skip out of the loop independent of whether the original loop condition still holds. This is represented by the **escape** component.

5 Implementation

This chapter discusses implementation details of the proposed solution. Section 5.1 discusses why the Rust Programming Language[9] was selected for the implementation of the BP engine. Section 5.2 introduces the interfaces of the DSL components introduced in Section 4.3. Section 5.3 discusses further the implemented HTTP Handler for the Protocol Handler component (see Figure 3), and the use of Signals within the implementation to realize different functionalities.

5.1 Why Rust?

Section 4.2 already introduced the criteria used for the selection of the programming language. This section discusses how these criteria led to the selection of the Rust programming language.

For criteria C1 and C3 compiled languages provide significant advantages compared to interpreted or hybrid languages. For C3 most interpreted and hybrid languages rely on an external runtime that needs to be supported on the target system. This additionally has significant performance issues (C1). The runtime environments of languages require additional memory and most popular languages that require an additional runtime (Java, C#, Ruby, Python) use a garbage collector mechanism to manage memory. Making memory management less deterministic and requiring even more compute capacity.

Thus the set of viable candidates is limited to compiled languages. Here Rust was selected due to criteria C2 and C4. Rust provides support for easily readable DSLs via Macros[10] and multi-line anonymous functions. Furthermore, there are examples of successful Rust DSLs such as SeaQuery[11] for formulating SQL queries in native Rust (C2). Additionally, in contrast to other performance-oriented languages such as C and C++, Rust uses the Ownership[12] memory management model instead of relying on the programmer to manage memory themselves. Thus, as long as the used libraries are implemented correctly (or just in safe Rust[13]) and no unsafe Rust code is used, a large

[9] `https://www.rust-lang.org/`, last access: 2025-01-01
[10] `https://doc.rust-lang.org/rust-by-example/macros/dsl.html`, last access: 2025-01-01
[11] `https://github.com/SeaQL/sea-query`, last access: 2025-01-01
[12] `https://doc.rust-lang.org/book/ch04-01-what-is-ownership.html`, last access: 2025-01-01
[13] `https://doc.rust-lang.org/nomicon/meet-safe-and-unsafe.html`, last access: 2025-01-01

class of memory-related issues such as data races[14] and most memory leaks[15] can be prevented. Additionally, safe Rust provides a memory safety guarantee [16] (C4).

5.2 DSL Implementation

The interface of the **call** and **manipulate** DSL components can be seen in Listing 1. Both are functions defined on an instance of the **Executive Component** (see Figure 3) which is called the Workflow Execution Engine Library (WEEL) in reference to [16]. Both take a unique identifier (id) and optional code artifacts. For the **manipulate** component, the specified code artifact implements the functionality. For the call the functionality is implemented by an external service. The URL of this service is stored within the execution context and is referenced within the model and the DSL solely by its assigned name. This is the *endpoint_name* within the call method signature. Further invocation Parameters are provided via the *parameters*. The different code parameters allow to specify handlers that can be utilized throughout different phases of the service invocation. Handler specified as *prepare_code* can be utilized to pre-process data elements that are passed to the service. These changes here are not persisted. *update_code* and *finalize_code* can be used to process the data returned by the service and change the state of the process model (data and endpoints). *rescue_code* can be used to handle exceptions when the external service call fails.

```
fn call(
    self: Arc<Self>,
    id: &str,
    endpoint_name: &str,
    parameters: HTTPParams,
    annotations: Value,
    prepare_code: Option<&str>,
    update_code: Option<&str>,
    finalize_code: Option<&str>,
    rescue_code: Option<&str>,
) -> Result<()>;

fn manipulate(self: Arc<Self>, id: &str, label: Option<&'static str>,
    code: Option<&str>) -> Result<()>;
```

Listing 1 *Interfaces for **call** and **manipulate** Activities*

The interface of the loop-related DSL components can be seen in Listing 2. **pre_test** and **post_test** take in the condition which is an expression (in this case in Ruby) and pass it on to the **loop_exec**'s *condition* parameter. The body of the loop is defined via the *lambda* parameter. It is the DSL representation of the process that is placed within the loop. The body is placed in an anonymous function that can be called an arbitrary number of times.

```
fn loop_exec(
    self: Arc<Self>,
    condition: [&str; 2],
    lambda: &(dyn Fn() -> Result<()> + Sync),
) -> Result<()>;
```

[14] https://blog.rust-lang.org/2015/04/10/Fearless-Concurrency.html, last access: 2025-01-01

[15] https://doc.rust-lang.org/book/ch15-06-reference-cycles.html, last access: 2025-01-01

[16] https://blog.rust-lang.org/2015/04/10/Fearless-Concurrency.html, last access: 2025-01-01

```
6
7  fn pre_test(condition: &str) -> [&str; 2];
8
9  fn post_test(condition: &str) -> [&str; 2];
```

Listing 2 *Interfaces for* ***loop_exec****,* ***pre_test****, and* ***post_test*** *Components*

The interface of the **parallel_do** and **parallel_branch** DSL components is provided in Listing 3. Here the *wait* counter can be provided to specify the number of branches that need to fulfill the condition provided by the *cancel* parameter for the remaining branches to be canceled. The *cancel* condition can either be First or Last for now. Thus a branch either fulfills the condition if the first activity or the last activity of the branch is done.

```
10 fn parallel_do(
11     self: Arc<Self>,
12     wait: Option<usize>,
13     cancel: CancelCondition,
14     lambda: &(dyn Fn() -> Result<()> + Sync + Send),
15 ) -> Result<()>;
16
17 fn parallel_branch(
18     self: Arc<Self>,
19     lambda: Arc<dyn Fn() -> Result<()> + Sync + Send>,
20 ) -> Result<()>;
```

Listing 3 *Interfaces for* ***parallel_do*** *and* ***parallel_branch*** *Components*

The interface of the **criticial_do** DSL component can be seen in Listing 4. It takes in the section of the process model it guards as the *lambda* parameter. It is an anonymous function where the body of the function is the DSL representation of the process model. Additionally, it takes in an identifier via the *mutex_id* that allows different sections of the process model, enclosed by multiple **critical_do** methods, to be handled as the same critical region when the *mutex_id* is the same. This ensures that all segments are guarded by the same mutex, ensuring mutual exclusion.

```
21     fn critical_do(self: Arc<Self>, mutex_id: &str,
22         lambda: &(dyn Fn() -> Result<()> + Sync)) -> Result<()>;
```

Listing 4 *Interface for the* ***criticial_do*** *Component*

The interfaces of the decision-related DSL components can be seen in Listing 5. Choose takes in whether the gateway is exclusive or inclusive via the *variant* parameter. The individual branches are defined within the *lambda* parameter as the body of the anonymous function in the order of evaluation as *alternative* function calls and an optional *otherwise* function call. The process model executed by the branches is defined as an anonymous function passed into the *lambda* parameter. The condition of the different alternative branches is passed in as the *condition* that is an expression formulated in a programming language (here only Ruby is supported for now).

```
23 fn choose(self: Arc<Self>, variant: ChooseVariant,
24     lambda: &(dyn Fn() -> Result<()> + Sync)) -> Result<()>;
25
26 fn alternative(self: Arc<Self>, condition: &str,
27     lambda: &(dyn Fn() -> Result<()> + Sync)) -> Result<()>;
```

```
28
29 fn otherwise(self: Arc<Self>,
30     lambda: &(dyn Fn() -> Result<()> + Sync)) -> Result<()>;
```

Listing 5 *Interfaces for the **choose**, **alternative**, and **otherwise** Components*

The interfaces for the **stop**, **terminate**, and **escape** DSL components can be seen in Listing 6. Their interfaces are trivial.

```
31 fn stop(self: Arc<Self>, id: &str) -> Result<()>;
32 fn terminate(self: Arc<Self>) -> Result<()>;
33 fn escape(self: Arc<Self>) -> Result<()>;
```

Listing 6 *Interfaces for the **stop**, **escape**, and **terminate** Components*

In conclusion, process models are translated into DSL components on a one-to-one basis. Nested process parts, e.g. branches in a parallel or decision gateway, or the body of a loop, are implemented as function calls wrapped in an anonymous function that can be executed an arbitrary number of times.

5.3 Artifact Implementation

This section discusses different interesting implementation details. The first section discusses the Protocol Handler of the HTTP protocol. The second section discusses Signals used within the implementation and the handler code.

HTTP Protocol Handler

The Protocol Handler implemented for this thesis's implementation supports HTTP. The main reason for this is that HTTP is the de-facto protocol of application layer protocol within the internet and with support of asynchronous call patterns as described by Mangler [23] and a bit discussed within the Signals section below allows for the implementation of Protocol Gateways for arbitrary other protocols such as OPC UA[17]. Thus satisfying requirement **R4** (see Section 4.1).

The handler is implemented on top of the Rust reqwest crate[18]. It provides a general-purpose interface to make HTTP requests and handle the returned results. For this it uses the *Parameter* enum as an exchange format with its callers. Their definition is provided in Listing 7. A **SimpleParameter** is used for simple key-value parameters, either within the requests/response body or as URL parameters

[17] `https://opcfoundation.org/about/opc-technologies/opc-ua/`, last access: 2025-01-01
[18] `https://crates.io/crates/reqwest`, last access: 2025-01-01

(for requests). This is controlled by the *param_type* field. A **ComplexParameter** is used for the transmission of files. These are always within the request/response body. For ComplexParameters, a mime type has to be provided via the *mime_type* field.

Key-value pairs and files can be added to an HTTP request by adding corresponding parameters to the client. The client will then construct an appropriate HTTP request. If only Key-value parameters are provided, the client will construct an `application/x-www-form-urlencoded` HTTP request, where the request body consists of the body parameters being URL-encoded and concatenated.

Given a set of parameters including a **ComplexParameter** the client will construct an HTTP Multipart Request with appropriate headers and field headers.

```
34 pub enum Parameter {
35     SimpleParameter {
36         name: String,
37         value: String,
38         param_type: ParameterType,
39     },
40     ComplexParameter {
41         name: String,
42         mime_type: Mime,
43         content_handle: File,
44     },
45 }
```

Listing 7 *Parameters Enum used as Exchange Format*

Signals

Signals are a messaging mechanism used within the process engine. Conceptually they signal the detection of certain events.

For one, if an external service sends an asynchronous update, with the necessary `cpee_update` header, then the `Signal::UpdateAgain` is raised within the engine. This signals the ConnectionWrapper to wait for further information from the external service.

Signals are also relevant for the Evaluation Service. Signals are used here to signal errors with the provided code (`Signal::SyntaxError, Signal::Error`) and for signaling the connection wrapper to retry the whole activity (including service invocations) (`Signal::Again`), to execute the rescue code (`Signal::Salvage`) or to stop the process (`Signal::Stop`).

5.4 Example

This section discusses concurrency aspects of the implementation based on an example process. This process is depicted in Figure 5. The transpiled process model code is provided in Listing 8. The following sections will describe how this process model is executed, focusing on concurrent execution aspects. A simplified UML sequence diagram of the interactions described below is provided in Figure 4. Here the activation bars of the mutex define how long the mutex is held by either the gateway thread or one of the branch threads.

Initially, the **parallel_do** will create data structures managed by the parallel gateway thread (gateway thread) to track the spawned branches. The reason for this is that the provided lambda could instantiate a dynamic number of branches. Thus the **parallel_do** does not yet know how many branches will be instantiated and needs to track this information dynamically. *Thread information* is stored in the *thread information map* which is guarded by a mutex to allow insertions into the map. Thus to access the thread information of any thread and modify it, the thread information map mutex has to be locked. Subsequently, a blocking queue (BQ) is instantiated to act as a barrier[19] that is used to synchronize the execution of the threads. Here the gateway code additionally instantiates a message channel (MC1) that is utilized later for thread communication. Both synchronization structures are managed by the *thread information* of the execution thread. Then the **thread information map mutex** is released to allow the parallel branch threads (branch threads) to access the data structures.

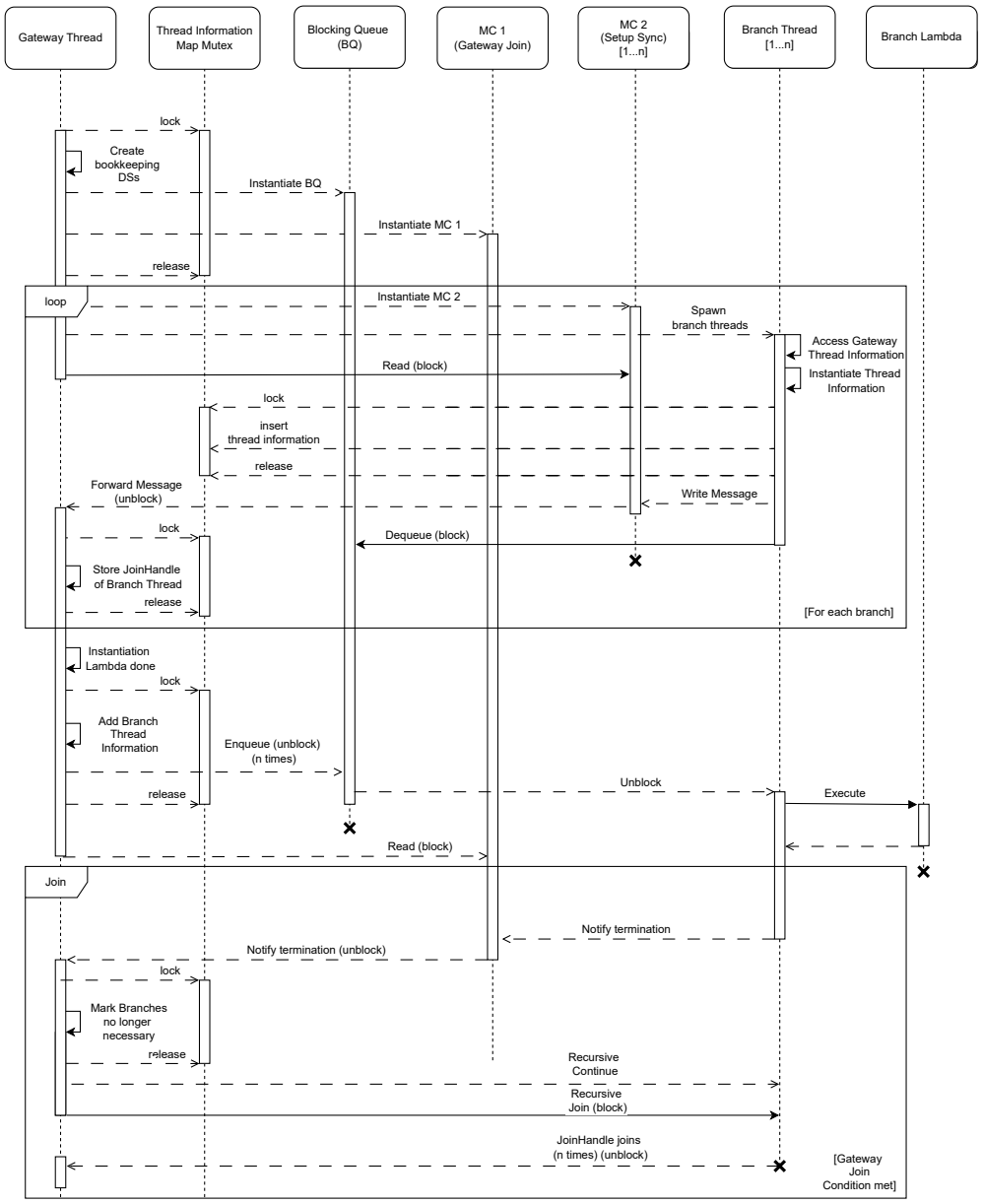
Subsequently, the provided lambda is executed to create the branches via the **parallel_branch** method calls. Note that all branches are thus spawned sequentially by the gateway thread. For each branch thread, a message channel (MC2) is instantiated to ensure that the new thread can run all necessary setup code before the gateway thread continues spawning the other threads. Then the gateway thread spawns the thread and waits for a message on MC2 before it attempts to lock the *thread information* of the branch thread to add the *JoinHandle*[20] of the branch thread.

Meanwhile, the branch thread instantiates its *thread information* and adds its *thread ID* to the data structure of the gateway thread. Thus the gateway branch now "knows" of the branch thread's existence. After the initial bookkeeping is done by the thread, a message on MC2 is published to

[19] `https://www.cs.utexas.edu/~rossbach/cs380p/lectures/06-Conditions+Barriers.pdf`, last access: 2025-01-01

[20] `https://doc.rust-lang.org/std/thread/struct.JoinHandle.html`, last access: 2025-01-01

Figure 4
Interaction of the Gateway and Branch Threads via Synchronization Structures



notify the gateway branch that it can continue execution. Then the branch thread waits at the barrier (BQ) for the gateway thread to signal that the branch can start execution of the lambda.

Once the parallel gateway thread finishes executing the provided lambda, and thus instantiates all branch threads, it locks its *thread information* mutex and populates it with further data necessary for the termination criteria specified. For this, the *thread information mutex* is locked. In the provided example, no *wait* counter is specified and thus the parallel gateway sets the cancel threshold to the number of spawned threads. Thus all threads will execute until termination.

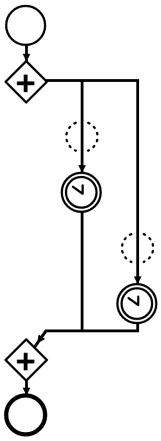
Then all branch threads are signaled to start execution via BQ. Thus both threads start executing the nested activities in parallel. Then the *thread information mutex* on the gateway thread is released.

After a branch thread finishes executing the provided lambda, it locks the *thread information map mutex* and accesses the gateway and its own *thread information*. Due to the locking of the thread information map only a single branch thread can enter the code section described in the following paragraph.

Then the number of terminated branch threads is incremented in the *thread information* of the gateway thread. Then the *thread information* of the branch thread is released to ensure that the subsequent code does not deadlock. Since the *CancelCondition* is set to last, the branch thread checks whether the threshold for termination is reached. In the example, this happens only in the final thread. If this is the case, sequentially the thread information of each branch thread is borrowed. All unfinished branches are then marked as no longer necessary and will skip the executing the remaining code and thus terminate nearly instantaneous (after it executes any active activity). Then the thread information reference of the branch thread is released to allow all of the child threads to continue recursively (necessary for nested parallel gateways). This means that activities are nudged to stop waiting and terminate. Then, independent of the *CancelCondition*, the branch thread checks whether the gateway termination condition is satisfied, and notifies the gateway via MC1 that it can execute the remaining bookkeeping code. Finally, it releases the *thread information map mutex*.

When the gateway thread gets notified via MC1 that the termination criterion is met, it will acquire the mutex on the *thread information map*. Then it sequentially borrows the *thread information* of

Figure 5
Example Process: Parallel Execution of Two Activities



each branch thread, marks it, if not terminated already, as no longer necessary, and releases the borrow *thread information* to recursively continues its branch threads. Lastly, the child threads are joined recursively, this ensures that all branch threads terminate before executing the gateway terminates.

```
46 weel!().parallel_do(None,Last,λ!({
47     weel!().parallel_branch(pλ!({
48         weel!().call(
49             "a1",
50             "timeout",
51             HTTPParams {
52                 label: "",
53                 method: Method::GET,
54                 arguments: json!([
55                         {
56                             "name": "timeout",
57                             "value": "2"
58                         }
59                 ]),
60             },
61             json!({
62                 ...
63             }),
64             None,
65             None,
66             None,
67             None,
68         )?;
69     }))?;
70     weel!().parallel_branch(pλ!({
71         weel!().call(
72             "a2",
73             "timeout",
74             HTTPParams {
75                 label: "",
76                 method: Method::GET,
77                 arguments: json!([
78                         {
79                             "name": "timeout",
80                             "value": "3"
81                         }
82                 ]),
83             },
84             json!({
85                 ...
86             }),
87             None,
88             None,
89             None,
90             None,
91         )?;
92     }))?;
93 }))?;
```

Listing 8 *Transpiled Example Process, Annotations replaced by . . . for brefity*

6 Evaluation

In this chapter, the resource efficiency of the proposed solution is measured against the reference implementation. While there are other attempts at resource-efficient process engines, such as [18]–[21] comparing against these quantitatively is challenging since all engines provide different feature sets and have different niches. Moreover, most papers provide no resource utilization measurements for their approaches (see Section 3.2). As such, to compare engines with equivalent feature sets, we decide to compare our solution against the CPEE reference implementation[21].

Section 6.1 will describe how the evaluation was conducted. It describes how the memory and execution time were measured and which processes were used for the experiments. Section 6.2 discusses the results of these experiments.

6.1 Evaluation Setup

For the evaluation, a set of process patterns (see Section 6.1) were modeled and executed using the reference implementation and the proposed implementation. In each model activities are a call to an external service that delays its response for a single second and then responds. For the proposed implementation a dynamically and a statically linked version was evaluated. Memory and execution time are measured for all patterns. For the memory analysis heap memory and Residual Set Size are considered. Residual Set Size (RSS) is measured for both implementations using the `time`[22]. To measure the heap allocations `heaptrack`[23] is used for the proposed implementation in Rust and `MemoryProfiler`[24] is used for the reference implementation in Ruby. For time measurements `time` is used for both implementations. For all measurements the time spent in the external service was not considered. Evaluation is done on two different platforms. To represent an unconstrained computing environment, a desktop computer with an AMD Ryzen 7 8745H and 32 GB of RAM (furthermore called **Echo**) is used. To represent a constrained computing device an Orange Pi Zero 2W[25] (furthermore called **Orange**) is used.

[21] CPEE Homepage
[22] `https://www.man7.org/linux/man-pages/man1/time.1.html` last access: 2025-01-01
[23] `https://github.com/KDE/heaptrack` last access: 2025-01-01
[24] `https://github.com/SamSaffron/memory_profiler` last access: 2025-01-01
[25] http://www.Orangepi.org/html/hardWare/computerAndMicrocontrollers/details/Orange-Pi-Zero-2W.html last access: 2025-01-01

Echo uses an evaluation service instance that is hosted on the machine. **Orange** utilizes an external evaluation service hosted within the internet and thus suffers from increased network delays. Echo is connected to the network via an Ethernet cable. Orange is connected via 2.4 GHz Wifi.

Set of Process Patterns

The set of process patterns consists of basic control flow patterns to ensure coverage and feature completeness of the benchmark. These patterns are:

- **Sequence**: A sequence of 3 activities
- **Parallel**: Parallel execution of 2 activities
- **Choose**: Alternative execution of 2 activities (only 1 is
- **Loop**: A loop that executes an activity a total of 3 times

While these patterns do not cover the full range of features supported by the DSL, they represent a foundational set of process components.

6.2 Results

The results of the experiments are depicted in multiple Tables. Table 4 shows the execution time measured for both implementations on both platforms. Here we can see that the proposed implementation performs better for every pattern on both platforms. For Echo, the proposed solution is more than 4 times faster for every pattern. This effect is even more pronounced on the low-compute Orange platform where the proposed implementation is faster by a factor of 5-13. The execution time measured for the proposed implementation increased by about one second when switching from the Echo to the Orange platform. For the Ruby implementation it increased by more than three seconds. This demonstrates that a compiled approach shows significant performance improvements, especially on resource-constrained platforms.

The results of the memory profiling for the experiments is depicted in Tables 5 and 6. Table 5 shows the measured Residual Set Size of both implementations across the different patterns. Table 6 shows the maximum heap size of both implementations during the execution of the different patterns. These measurements show that the proposed implementation uses significantly less heap, by a factor of about 6-10 depending on the pattern across all platforms.

Table 4
Time Measurements

Pattern	Rust				Ruby	
	Time **Echo**		Time **Orange**		Time **Echo**	Time **Orange**
	Static	Dynamic	Static	Dynamic		
Sequence	0.25 s	0.26 s	1.12 s	1.42 s	1.35 s	6.27 s
Parallel	0.09 s	0.10 s	0.40 s	0.42 s	1.16 s	5.42 s
Choose	0.09 s	0.10 s	0.73 s	0.51 s	1.32 s	5.59 s
Loop	0.27 s	0.30 s	1.31 s	1.47 s	1.49 s	6.69 s

The measured Residual Set Size of the proposed solution is also significantly smaller than for the proposed solution (see Table 5). The statically linked approach has an RSS that is smaller by a factor of more than 10 across both platforms. The dynamically linked approach by a factor of about 5. However here it is important to mention that these factors are a lower bound if multiple instances of the same program are run. The reason for this is that for each instance an individual ruby interpreted has to be instantiated and loaded into memory, consuming a significant amount of memory. For the dynamically linked version of the proposed solution this does not happen, the standard library code and the complete application code are linked dynamically, and thus only need to be kept in memory once. Each process instance is thus very lightweight. Therefore, given that multiple instances are executed on the same platform, significantly less memory is used by the proposed implementation.

When comparing the memory consumption of the proposed solution with the implementations discussed in the related work (see Chapter 3) then it becomes clear that the implementation also uses less than 22 MB of memory like Sliver. Furthermore, the statically linked implementation uses less memory than the ERWF engine that provides no network connectivity, but real-time support. Thus the proposed solution shows significant memory efficiency compared to related engines and compared to the reference implementation (**R1**).

Table 5
Memory Measurements (RSS)

Pattern	Rust				Ruby	
	Memory **Echo**		Memory **Orange**		Memory **Echo**	Memory **Orange**
	Static	Dynamic	Static	Dynamic		
Sequence	6.78 MiB	14.00 MiB	5.82 MiB	13.60 MiB	94.5 MiB	85.0 MiB
Parallel	7.86 MiB	15.33 MiB	7.10 MiB	14.82 MiB	94.5 MiB	85.3 MiB
Choose	6.67 MiB	14.25 MiB	6.18 MiB	13.87 MiB	94.0 MiB	85.3 MiB
Loop	6.88 MiB	14.53 MiB	6.42 MiB	14.28 MiB	94.3 MiB	86.4 MiB

Table 6
Memory Measurements (Heap)

Pattern	Rust				Ruby	
	Memory **Echo**		Memory **Orange**		Memory **Echo**	Memory **Orange**
	Static	Dynamic	Static	Dynamic		
Sequence	619.1 KiB	619.2 KiB	598.6 KiB	598.6 KiB	5790.0 KiB	5790.0 KiB
Parallel	966.1 KiB	977.4 KiB	946.5 KiB	799.1 KiB	5800.0 KiB	5810.0 KiB
Choose	949.9 KiB	950.0 KiB	982.3 KiB	982.4 KiB	6850.0 KiB	6850.0 KiB
Loop	998.5 KiB	998.3 KiB	1000.0 KiB	1000.0 KiB	6840.0 KiB	6850.0 KiB

7 Discussion

To address the lack of computing resources on IoT devices (see **C3** in Section 2.2) for IoT-enhanced BPM this thesis aimed to answer 3 core research questions:

RQ 1 What are the challenges for IoT-enhanced BPM to optimize for CPU and memory utilization?

RQ 2 How can a resource-efficient Process Engine be implemented?

RQ 3 Does the Process engine implemented for **RQ 2** fulfill the identified requirements?

Research Question 1 (**RQ 1**) was answered by conducting a literature review in Chapter 3. Here relevant works in the area of efficient process execution were analyzed and discussed (see Section 3.2). This Chapter also identified a research gap that this thesis attempted to solve (see Section 3.3).

For Research Question 2 (**RQ 2**) Chapter 4 proposed a solution. For this, based on the related works and the identified gap, requirements were identified (see Section 4.1):

R1 Efficient utilization of computing resources by the engine

R2 Provide Logging for monitoring and (future: recovery (see **C4** in Section 2.2)

R3 Development of a powerful, but high level Process DSL

R4 Support for HTTP, including HTTP multipart

The general architecture of the solution is outlined in Section 4.2. Which includes a model-to-code approach for efficient process execution (**R1**) and complete logging capabilities (**R2**). To complement the model-to-code approach a powerful internal DSL was proposed in Sections 4.3, and 5.2 (**R2, R3**). For the host language, Rust was selected based on the criteria discussed in Section 4.2. The reasoning behind selecting Rust was discussed in Section 5.1. The implemented HTTP Protocol Handler, which handles all communication with external services, was discussed in Section 5.3 (**R4**).

To answer Research Question 3, and thus to know whether requirement **R1** is satisfied, an evaluation was conducted and discussed in Section 6. For this, a set of basic process patterns were modeled and executed on two platforms, one representing a resource-constrained, and the other one a resource-unconstrained computing platform. For both implementations and platforms speed and memory

consumption were measured. Across the board, the proposed solution shows significantly more efficient use of computing resources. The evaluation also shows that the engine is more memory efficient than most of the related engines, and consumes significantly less than the threshold described in the paper that proposed the Sliver engine [19] that provides comparative features. Thus this thesis provides a novel solution to the field of efficient process execution and allows the execution of process models on resource-constrained platforms. Thus enabling Mist Computing architectures for IoT-enhanced BPM.

8 Conclusion

Internet of Things (IoT) technology has been widely adopted across many domains, such as manufacturing and healthcare. IoT offers multiple advantages such as improved data quantity, quality, and recentness for decision-making and analysis processes. Nonetheless the application of IoT introduces many architectural and operational challenges, such as the heterogeneous nature of the different interfaces and capabilities of devices, scalability, unreliable power supply and network connectivity.

IoT-enhanced BPM aims to harness the benefits of IoT technology for Business Process Management while addressing its architectural and operational challenges. To get closer to this goal, this thesis developed a resource-efficient process engine based on the Cloud Process Execution Engine (CPEE) [17], [23]. The engine demonstrates that process models can be efficiently executed using a model-to-code approach while keeping the generated code maintainability and easily understandable by leveraging a Process DSL [16]. This allows the execution of general-purpose process models on resource-constrained platforms such as the Orange Pi Zero 2W[26].

Nonetheless, there are still open challenges for the execution of process models within IoT environments. While the logging can provide fault tolerance, no mechanism was implemented to handle disconnects when talking to external services or the middleware component (redis[27]). Furthermore, the mentioned possibility of recovery after a crash, e.g., due to power supply issues, common in an IoT environment, is not yet implemented. Additionally the implemented Protocol Handler relies on the `reqwest` crate[28] that requires the heavy-weight async tokio runtime[29], which in hindsight is not beneficial. Thus implementing a protocol handler on a more lightweight http crate would provide significant performance savings. Another aspect where the current implementation is suboptimal as it utilizes the regex crate[30] for extracting information out of structured callbacks from the BPMS that consumes a major chunk of the total memory (about 80% of peak heap memory consumption). Replacing both dependencies would significantly decrease the memory consumption. Lastly, the

[26] `http://www.Orangepi.org/html/hardWare/computerAndMicrocontrollers/details/Orange-Pi-Zero-2W.html`, last access: 2025-01-01
[27] `https://redis.io/` last access: 2025-01-01
[28] `https://crates.io/crates/reqwest` last access: 2025-01-01
[29] `httpss://crates.io/crates/tokio` last access: 2025-01-01
[30] `https://crates.io/crates/regex` last access: 2025-01-01

implementation relies on the Rust Runtime[31] for the `std` crate[32], and thus requires a memory allocator to function.

[31] `https://doc.rust-lang.org/reference/runtime.html` last access: 2025-01-01
[32] `https://doc.rust-lang.org/std/` last access: 2025-01-01

Bibliography

[1] C. Chang, S. N. Srirama, and R. Buyya, "Mobile cloud business process management system for the internet of things: A survey," *ACM Computing Surveys*, vol. 49, no. 4, pp. 1–42, Dec. 2016, ISSN: 1557-7341. DOI: 10.1145/3012000.

[2] J. Mangler, R. Seiger, J.-V. Benzin, *et al.*, *From internet of things data to business processes: Challenges and a framework*, 2024. DOI: 10.48550/ARXIV.2405.08528.

[3] C. Janiesch, A. Koschmider, M. Mecella, *et al.*, "The internet of things meets business process management: A manifesto," *IEEE Systems, Man, and Cybernetics Magazine*, vol. 6, no. 4, pp. 34–44, Oct. 2020, ISSN: 2380-1298. DOI: 10.1109/msmc.2020.3003135.

[4] M. Sneps-Sneppe and D. Namiot, "On web-based domain-specific language for internet of things," in *2015 7th International Congress on Ultra Modern Telecommunications and Control Systems and Workshops (ICUMT)*, IEEE, Oct. 2015, pp. 287–292. DOI: 10.1109/icumt.2015.7382444.

[5] C. Chang, S. N. Srirama, and R. Buyya, "Internet of things (iot) and new computing paradigms," 2018. DOI: 10.48550/ARXIV.1812.00591.

[6] M. Dumas, M. La Rosa, J. Mendling, and H. A. Reijers, *Fundamentals of Business Process Management*. Springer Berlin Heidelberg, 2018, ISBN: 9783662565094. DOI: 10.1007/978-3-662-56509-4.

[7] J. Mass, C. Chang, and S. N. Srirama, "Context-aware edge process management for mobile thing-to-fog environment," in *Proceedings of the 12th European Conference on Software Architecture: Companion Proceedings*, ser. ECSA '18, ACM, Sep. 2018. DOI: 10.1145/3241403.3241449.

[8] J. Mass, S. N. Srirama, and C. Chang, "Step-one: Simulated testbed for edge-fog processes based on the opportunistic network environment simulator," *Journal of Systems and Software*, vol. 166, p. 110 587, Aug. 2020, ISSN: 0164-1212. DOI: 10.1016/j.jss.2020.110587.

[9] S. W. Loke, "Service-oriented device ecology workflows," in *Service-Oriented Computing - ICSOC 2003*. Springer Berlin Heidelberg, 2003, pp. 559–574, ISBN: 9783540245933. DOI: 10.1007/978-3-540-24593-3_38.

[10] F. De Luzi, F. Leotta, A. Marrella, and M. Mecella, "On the interplay between business process management and internet-of-things: A systematic literature review," *Business & Information Systems Engineering*, Mar. 2024, ISSN: 1867-0202. DOI: 10.1007/s12599-024-00859-6.

[11] S. Schönig, L. Ackermann, and S. Jablonski, "Internet of things meets bpm: A conceptual integration framework," in *Proceedings of 8th International Conference on Simulation and Modeling Methodologies, Technologies and Applications*, SCITEPRESS - Science and Technology Publications, 2018, pp. 307–314. DOI: 10.5220/0006824803070314.

[12] J. Mass, C. Chang, and S. N. Srirama, "Edge process management: A case study on adaptive task scheduling in mobile iot," *Internet of Things*, vol. 6, p. 100 051, Jun. 2019, ISSN: 2542-6605. DOI: 10.1016/j.iot.2019.100051.

[13] C. Stoiber and S. Schönig, "Event-driven business process management enhancing iot – a systematic literature review and development of research agenda," in *Innovation Through Information Systems*. Springer International Publishing, 2021, pp. 645–661, ISBN: 9783030868000. DOI: 10.1007/978-3-030-86800-0_44.

[14] Hevner, March, Park, and Ram, "Design science in information systems research," *MIS Quarterly*, vol. 28, no. 1, p. 75, 2004, ISSN: 0276-7783. DOI: 10.2307/25148625.

[15] P. P.-S. Chen, "The entity-relationship model—toward a unified view of data," *ACM Transactions on Database Systems*, vol. 1, no. 1, pp. 9–36, Mar. 1976, ISSN: 1557-4644. DOI: 10.1145/320434.320440.

[16] G. Sturmer, J. Mangler, and E. Schikuta, "A domain specific language and workflow execution engine to enable dynamic workflows," in *2009 IEEE International Symposium on Parallel and Distributed Processing with Applications*, IEEE, 2009, pp. 653–658. DOI: 10.1109/ispa.2009.106.

[17] J. Mangler, G. Stuermer, and E. Schikuta, "Cloud process execution engine - evaluation of the core concepts," University of Vienna, Facultiy of Computer Science Workflow Systems and Technologies Group, Tech. Rep., 2010. DOI: https://doi.org/10.48550/arXiv.1003.3330.

[18] L. Pajunen and S. Chande, "Developing workflow engine for mobile devices," in *11th IEEE International Enterprise Distributed Object Computing Conference (EDOC 2007)*, IEEE, Oct. 2007, pp. 279–279. DOI: 10.1109/edoc.2007.32.

[19] G. Hackmann, M. Haitjema, C. Gill, and G.-C. Roman, "Sliver: A bpel workflow process execution engine for mobile devices," in *Service-Oriented Computing – ICSOC 2007*. Springer Berlin Heidelberg, 2006, pp. 503–508, ISBN: 9783540749745. DOI: 10.1007/11948148_47.

[20] T. Chou, S. Chang, Y. Lu, *et al.*, "Emwf for flexible automation and assistive devices," in *2009 15th IEEE Real-Time and Embedded Technology and Applications Symposium*, IEEE, Apr. 2009, pp. 243–252. DOI: 10.1109/rtas.2009.21.

[21] M. S. J. Schobel R. Pryss and M. Reichert, "A lightweight process engine for enabling advanced mobile applications," in *On the Move to Meaningful Internet Systems: OTM 2016 Conferences*, C. Debruyne, ·. Hervé, P. Robert, *et al.*, Eds., Springer International Publishing, 2016, pp. 552–569, ISBN: 9783319484723. DOI: 10.1007/978-3-319-48472-3_33.

[22] A. Lindsay, D. Downs, and K. Lunn, "Business processes—attempts to find a definition," *Information and Software Technology*, vol. 45, no. 15, pp. 1015–1019, Dec. 2003, ISSN: 0950-5849. DOI: 10.1016/s0950-5849(03)00129-0.

[23] J. Mangler and S. Rinderle-Ma, "Cloud process execution engine: Architecture and interfaces," Department of Informatics, Technical University of Munich, Tech. Rep., 2022. DOI: https://doi.org/10.48550/arXiv.2208.12214.

[24] G. Aagesen and J. Krogstie, "Bpmn 2.0 for modeling business processes," in *Handbook on Business Process Management 1*. Springer Berlin Heidelberg, Apr. 2014, pp. 219–250, ISBN: 9783642451003. DOI: 10.1007/978-3-642-45100-3_10.

[25] J. Nitzsche, T. van Lessen, D. Karastoyanova, and F. Leymann, "Bpel for semantic web services (bpel4sws)," in *On the Move to Meaningful Internet Systems 2007: OTM 2007 Workshops*. Springer Berlin Heidelberg, 2007, pp. 179–188, ISBN: 9783540768883. DOI: 10.1007/978-3-540-76888-3_37.

[26] M. Reichert and B. Weber, *Enabling Flexibility in Process-Aware Information Systems*. Springer Berlin Heidelberg, 2012, ISBN: 9783642304095. DOI: 10.1007/978-3-642-30409-5.

[27] K. Ashton, "That 'internet of things' thing," 2009.

[28] J. Mangler, J. Grüger, L. Malburg, *et al.*, "Datastream xes extension: Embedding iot sensor data into extensible event stream logs," *Future Internet*, vol. 15, no. 3, p. 109, Mar. 2023, ISSN: 1999-5903. DOI: 10.3390/fi15030109.

[29] H. Kopetz and W. Steiner, *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Springer International Publishing, 2022, ISBN: 9783031119927. DOI: 10.1007/978-3-031-11992-7.

[30] M. J. Martin, "Cloud, fog, and now, mist computing," 2015. [Online]. Available: https://www.linkedin.com/pulse/cloud-computing-fog-now-mist-martin-ma-mba-med-gdm-scpm-pmp.

[31] M. Satyanarayanan, "The emergence of edge computing," *Computer*, vol. 50, no. 1, pp. 30–39, Jan. 2017, ISSN: 0018-9162. DOI: 10.1109/mc.2017.9.

[32] F. Bonomi, R. Milito, J. Zhu, and S. Addepalli, "Fog computing and its role in the internet of things," in *Proceedings of the first edition of the MCC workshop on Mobile cloud computing*, ser. SIGCOMM '12, ACM, Aug. 2012. DOI: 10.1145/2342509.2342513.

[33] Y. Chen and G. D. Luca, "Technologies for developing a smart city in computational thinking," *International Journal of Simulation and Process Modelling*, vol. 13, no. 2, p. 91, 2018, ISSN: 1740-2131. DOI: 10.1504/ijspm.2018.091742.

[34] M. Mernik, J. Heering, and A. M. Sloane, "When and how to develop domain-specific languages," *ACM Computing Surveys*, vol. 37, no. 4, pp. 316–344, Dec. 2005, ISSN: 1557-7341. DOI: 10.1145/1118890.1118892.

[35] L. M. Do Nascimento, D. L. Viana, P. Neto, D. Martins, V. C. Garcia, and S. Meira, "A systematic mapping study on domain-specific languages," in *The Seventh International Conference on Software Engineering Advances (ICSEA 2012)*, 2012, pp. 179–187.

[36] M. Fowler and R. Parsons, *Domain Specific Languages*. Addison-Wesley Professional, 2010, ISBN: 9780132107549.

[37] R. Sen, G.-C. Roman, and C. Gill, "Cian: A workflow engine for manets," in *Coordination Models and Languages*. Springer Berlin Heidelberg, 2008, pp. 280–295, ISBN: 9783540682653. DOI: 10.1007/978-3-540-68265-3_18.

[38] W.-C. Chen and C.-S. Shih, "Erwf: Embedded real-time workflow engine for user-centric cyber-physical systems," in *2011 IEEE 17th International Conference on Parallel and Distributed Systems*, IEEE, Dec. 2011, pp. 713–720. DOI: 10.1109/icpads.2011.60.

[39] R. Pryss, J. Tiedeken, U. Kreher, and M. Reichert, "Towards flexible process support on mobile devices," in *Information Systems Evolution*. Springer Berlin Heidelberg, 2011, pp. 150–165, ISBN: 9783642177224. DOI: 10.1007/978-3-642-17722-4_11.

[40] N. Glombitza, S. Ebers, D. Pfisterer, and S. Fischer, "Using bpel to realize business processes for an internet of things," in *Ad-hoc, Mobile, and Wireless Networks*. Springer Berlin Heidelberg, 2011, pp. 294–307, ISBN: 9783642224508. DOI: 10.1007/978-3-642-22450-8_23.

[41] C. Chang, S. N. Srirama, and S. Ling, "Spica: A social private cloud computing application framework," in *Proceedings of the 13th International Conference on Mobile and Ubiquitous Multimedia*, ser. MUM '14, ACM, Nov. 2014, pp. 30–39. DOI: 10.1145/2677972.2677979.

[42] C. Chang, S. N. Srirama, and J. Mass, "A middleware for discovering proximity-based service-oriented industrial internet of things," in *2015 IEEE International Conference on Services Computing*, IEEE, Jun. 2015. DOI: 10.1109/scc.2015.27.

[43] C. Chang, S. N. Srirama, and S. Ling, "An adaptive mediation framework for mobile p2p social content sharing," in *Service-Oriented Computing*. Springer Berlin Heidelberg, 2012, pp. 374–388, ISBN: 9783642173585. DOI: 10.1007/978-3-642-34321-6_25.

[44] D. Uckelmann, M. Harrison, and F. Michahelles, "An architectural approach towards the future internet of things," in *Architecting the Internet of Things*. Springer Berlin Heidelberg, 2011, pp. 1–24, ISBN: 9783642191572. DOI: 10.1007/978-3-642-19157-2_1.

[45] A. Ernest, E. Mensah, and A. Gilbert, "Qualitative assessment of compiled, interpreted and hybrid programming languages," *Communications on Applied Electronics*, vol. 7, no. 7, pp. 8–13, Oct. 2017, ISSN: 2394-4714. DOI: 10.5120/cae2017652685.