Institute of Software Engineering
Software Quality and Architecture

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Master's Thesis

Towards Understanding the Energy Efficiency of Different Software Architecture Styles in Cloud-Based Environments: A Case Study Comparing Microservices and Modular Monolith Based on Two Variants of an Example System

David Kopp

Course of Study:	Softwaretechnik
Examiner:	Prof. Dr.-Ing. Steffen Becker
Supervisor:	Prof. Dr.-Ing. Steffen Becker, Dipl.-Inf. Peter Kutschera
Commenced:	September 29, 2023
Completed:	March 29, 2024

Abstract

Context. Currently, overcoming the climate crisis is a major and essential task for us humans, to which the software industry can and must also make a contribution. This thesis deals with possibilities to reduce the energy consumption caused by software, which is one aspect of Green Software Engineering.

Problem. Software can require different amounts of energy for the same functionality, depending on how it is implemented and operated. This has received relatively little attention, partly because of a lack of awareness and knowledge. There is a need for more transparency and simple methods to measure the energy efficiency of software applications. There is also a lack of concrete recommendations for practical action on how to design and implement energy-efficient software.

Objective. The objective of this thesis is to develop recommendations for architectural decisions for cloud-based web applications with the aim of improving energy efficiency. For this purpose, the two backend architecture styles microservices and modular monolith are compared with each other.

Method. First, two applications with the same functionality are provided, one in a microservices architecture and one as a modular monolith. Then, measurements are performed with the Green Metrics Tool in an isolated measurement environment and with Kepler in a Kubernetes cluster to draw conclusions about the energy efficiency of the two architectural styles.

Result. The results of the measurements with the Green Metrics Tool show that the monolithic reference system performs significantly better in simple scenarios. This is mainly due to higher response times in the microservices system as a result of network communication and data processing. Microservices have advantages as soon as scaling is required. Unfortunately, no representative measurements could be carried out with Kepler within the time frame of this thesis, so no concrete statements can yet be made in this regard.

Conclusion. The scalability requirement of a software system plays a critical role in determining optimal architectural decisions for maximum energy efficiency. The pursuit of unlimited scalability can potentially negatively impact performance and energy efficiency when it is not needed at all. However, in scenarios with dynamic and high loads that require efficient scalability, fine-grained architectures such as microservices offer advantages. The migration from a microservices architecture to a modular monolith has also shown that modularity can be realized in both architectural styles, but with less complexity in the monolithic system.

Kurzfassung

Kontext. Die Bewältigung der Klimakrise ist derzeit eine große und essentielle Aufgabe für uns Menschen, zu der auch die Softwareindustrie einen Beitrag leisten kann und muss. Diese Arbeit befasst sich mit Möglichkeiten, den durch Software verursachten Energieverbrauch zu reduzieren, was einen Aspekt von Green Software Engineering darstellt.

Problemstellung. Software kann für dieselbe Funktionalität unterschiedlich viel Energie benötigen, je nachdem, wie sie implementiert und betrieben wird. Diesem Aspekt wurde bisher relativ wenig Aufmerksamkeit geschenkt, was zum Teil auf einen Mangel an Bewusstsein und Wissen zurückzuführen ist. Es besteht ein Bedarf an mehr Transparenz und einfachen Methoden zur Messung der Energieeffizienz von Softwareanwendungen. Außerdem fehlt es an konkreten Handlungsempfehlungen für die Praxis, wie energieeffiziente Software entworfen und umgesetzt werden kann.

Zielsetzung. Ziel dieser Arbeit ist es, Empfehlungen für Architekturentscheidungen für Cloud-basierte Webanwendungen zu entwickeln, die eine Verbesserung der Energieeffizienz bewirken sollen. Zu diesem Zweck werden die beiden Backend-Architekturstile Microservices und modularer Monolith miteinander verglichen.

Methode. Zunächst werden zwei Anwendungen mit der gleichen Funktionalität bereitgestellt, eine in einer Microservices-Architektur und eine als modularer Monolith. Anschließend werden Messungen mit dem Green Metrics Tool in einer isolierten Messumgebung und mit Kepler in einem Kubernetes-Cluster durchgeführt, um Rückschlüsse auf die Energieeffizienz der beiden Architekturstile zu ziehen.

Ergebnis. Die Ergebnisse der Messungen mit dem Green Metrics Tool zeigen, dass das monolithische Referenzsystem in einfachen Szenarien deutlich besser abschneidet. Dies liegt vor allem an den höheren Antwortzeiten im Microservices-System, die durch die Netzwerkkommunikation und die Datenverarbeitung bedingt sind. Microservices sind im Vorteil, sobald eine Skalierung erforderlich ist. Leider konnten im Zeitrahmen dieser Arbeit keine repräsentativen Messungen mit Kepler durchgeführt werden, so dass diesbezüglich noch keine konkreten Aussagen getroffen werden können.

Schlussfolgerung. Die Anforderung an die Skalierbarkeit eines Softwaresystems spielt eine entscheidende Rolle bei der Bestimmung optimaler Architekturentscheidungen für maximale Energieeffizienz. Das Streben nach unbegrenzter Skalierbarkeit kann sich potenziell negativ auf die Leistung und Energieeffizienz auswirken, wenn sie gar nicht benötigt wird. In Szenarien mit dynamischen und hohen Lasten, die eine effiziente Skalierbarkeit erfordern, bieten feingranulare Architekturen wie Microservices jedoch Vorteile. Die Migration von einer Microservices-Architektur zu einem modularen Monolithen hat zudem gezeigt, dass Modularität in beiden Architekturstilen realisiert werden kann, allerdings mit weniger Komplexität im monolithischen System.

Acknowledgement

I would particularly like to thank Peter Kutschera, Konrad Pfeilsticker and Prof. Steffen Becker. Without them, this work would not have been possible.

Thank you Peter for supporting me so intensively and always believing in me, even when I had mentally difficult phases.
Thank you Konrad for paving the way for me to find a suitable topic for this Master's thesis in collaboration with the envite consulting GmbH.
Thank you Steffen for supervising me, spending much time on many meetings, for giving me a lot of freedom and allowing me to work on a topic of my own choice in the field of sustainable software development.

I would also like to thank Arne Tarara for allowing me to use the measurement cluster of the Green Metrics Tool free of charge and for providing helpful insights in some discussions about how to do energy measurements. It was also a great pleasure to work with Arne on a few small code changes to the Green Metrics Tool.

Contents

List of Figures

List of Tables

List of Listings

Acronyms

ACID Atomicity, Consistency, Isolation, Durability. 13

API Application Programming Interface. 10

AWS Amazon Web Services. 7

CCF Cloud Carbon Footprint. 25

CDC Change Data Capture. 45

CDN Content Delivery Network. 26

CI Continouos Integration. 32

CLI Command Line Interface. 60

CPU Central Processing Unit. 21

DDD Domain-Driven Design. 9

DSLAM Digital Subscriber Line Access Multiplexer. 21

EE Enterprise Edition. 3

EKS Elastic Kubernetes Service. 52

FOSS Free and Open Source Software. 66

FTP File Transfer Protocol. 65

GHG Greenhouse Gas. 1

GMT Green Metrics Tool. 32

GSF Green Software Foundation. 16

GSMM Green Software Measurement Model. 32

HATEOAS Hypermedia as the Engine of Application State. 43

HDD Hard Disk Drive. 25

HPA Horizontal Pod Autoscaler. 56

HTTP Hypertext Transfer Protocol. 40

ICT Information and Communication Technologies. 1

IEA International Energy Agency. 1

IETF Internet Engineering Task Force. 22

IP Internet Protocol. 21

IPCC Intergovernmental Panel on Climate Change. 1

ISO International Organization for Standardization. 26

JDBC Java Database Connectivity. 65

JIT Just In Time. 76

JMX Java Management Extensions. 65

JSON JavaScript Object Notation. 59

JSP Jakarta Server Pages. 42

JVM Java Virtual Machine. 12

LoC Lines of Code. 59

LTS Long-Term Support. 42

MSE mean squared error. 31

NIST National Institute of Standards and Technology. 6

OS Operating System. 29

PUE Power Usage Effectiveness. 19

RAPL Running Average Power Limit. 28

REST Representational State Transfer. 3

SCI Software Carbon Intensity. 26

SDG Sustainable Development Goal. 5

SLO Service Level Objective. 39

SPC Storage Performance Council. 31

SPEC Standard Performance Evaluation Corporation. 31

SSD Solid Disk Drive. 25

TPC Transaction Processing Performance Council. 31

UI User Interface. 41

UML Unified Modeling Language. 8

URL Uniform Resource Locator. 52

vCPU virtual Central Processing Unit. 15

VM Virtual Machine. 30

XML Extensible Markup Language. 65

1 Introduction

Information and Communication Technologies (ICT), including TVs and other consumer electronics, are currently responsible for around 2.1–3.9 % of Greenhouse Gas (GHG) emissions and thus contribute a significant portion to the further progress of man-made climate change [FBW+21]. The widespread negative impacts of climate change on nature and on humans are serious and threaten the livelihoods of more than three billion people, as the current *Sixth Assessment Report* of the Intergovernmental Panel on Climate Change (IPCC) impressively illustrates [LCD+23]. In the face of this dramatic situation, it is essential that the ICT sector also takes its responsibility to reduce greenhouse gas emissions to zero as quickly as possible. Around 30 % of the emissions are coming from *embodied emissions*, which arise from the manufacture, transportation and disposal of ICT devices [FBW+21]. The largest share is accounted during usage through the consumption of energy.

Data centers play a major role in this. Data centers, excluding cryptocurrency mining, are estimated to consume 240–340 TWh of energy globally in 2022, representing 1–1.3 % of global energy demand [IEA23]. Since 2015, the total amount consumed by data centers worldwide has increased by 20–70 %. This is a moderate increase given the strong growth in workloads, largely as a result of major efficiency improvements by large data center operators in recent years. [MSL+20]. However, it must be expected that global energy consumption by data centers will continue to increase in the future. The International Energy Agency (IEA) even fears that the energy consumption of data centers could double between 2022 and 2026 due to the increasing demand for digital services, the rapid growth in the field of artificial intelligence and the further growth of cryptocurrency [IEA24].

1.1 Problem Statement

Software contributes significantly to GHG emissions in the ICT sector, not directly, but indirectly through the hardware resources required (embodied carbon) and the influence on the utilization of hardware components, which ultimately leads to the consumption of electrical energy.

There are various ways to reduce software-related emissions. These include, among others:

- Adjustments to business requirements (e.g. reducing the accuracy of computing operations)
- Increasing efficiency (energy and resource efficiency, e.g. by improving an algorithm)
- Implementation of carbon awareness strategies (e.g. execution of computing operations when a lot of solar energy is available)
- Long-term support for older hardware (counteracting software-related obsolescence)

This shows that there are a number of opportunities for stakeholders in the software development process and throughout the software lifecycle to contribute to climate protection.

However, there is often either a lack of awareness or a lack of information to implement appropriate measures towards more sustainable software, as many studies illustrate [MBZ+16; ORRP20; PC17; PHAH16]. In science, the aspect of energy efficiency in particular has received significant attention in the last roughly 10 years [CMG+20; LKM+13]. For example, there are studies on the energy efficiency of programming languages, programming techniques, design patterns and measurement approaches. However, there are still comparatively few studies that deal with the intersection between sustainability and software architecture [ABPL22]. This is problematic because, as mentioned from Andrikopoulos et al., architectural practices enable the reasoning and evaluation of sustainability as a quality attribute over the course of the software lifecycle. In addition, in daily software architecture work in particular, it is essential to know best practices and have supporting information available [FL23]. In many existing studies of software energy efficiency, the motivation has been to extend battery life on mobile devices. The consideration of the topic in the area of cloud computing is still very new and has only become more attractive in science in recent years [VLVH22]. More research is needed to understand how software architecture decisions in cloud-native applications affect energy efficiency and the carbon footprint.

It needs data in order to identify potential for optimization and make targeted improvements in the software development process. However, measuring both energy consumption and GHG emissions is associated with challenges. Carbon accounting, i.e. calculating the GHG emissions generated in a specific period at a specific location, is difficult [CB19]. In practice, estimates are therefore used for this, based on various parameters and proxy metrics.

A special characteristic of modern cloud-native applications is that they often consist of many or even thousands of distributed software components. Such fine-grained architectures promise, among other things, that they can guarantee good horizontal scalability, development teams can make and deploy changes quickly and the overall benefits of cloud computing can be utilized in the best possible way.

There are also various recommendations on software architecture for cloud-native applications with regard to energy efficiency and reducing the carbon footprint, for example from the major public cloud providers. However, there are hardly any concrete studies to date that provide figures on this, making it difficult to make evidence-based decisions.

1.2 Solution Approach

This thesis contributes to this by comparing the energy efficiency of two architectural styles for the backend of cloud-based web applications, which differ in particular in the degree of distribution: *modular monolith* and *microservices*.

A modular monolith is provided as a single complete application, i.e. there is only one deployment artefact, but is internally structured in a modular manner. Microservices is an architectural style that is distributed and each module is an independent deployed service with well-defined interfaces. In the sense of *domain-driven design*, both architecture styles can be structured based on the same bounded contexts and domain-specific building blocks.

For the study, a reference application is provided in two variants, one as a modular monolith and one in a microservices architecture. The two variants differ only in their architecture and the resulting differences; the functionality and also the degree of modularity in the sense of domain-driven design are identical for both. The two variants are then compared in various experiments and with different methods in terms of their energy efficiency.

Energy efficiency is examined instead of the carbon footprint so as not to rely purely on estimates with many parameters, but also to be able to carry out concrete measurements.

In order for the software industry as a whole to become more sustainable and environmentally friendly, however, it is essential in the medium term not only to look at energy consumption, but also to include other aspects such as climate, water, e-waste, etc., to make the effects transparent and to take action.

1.3 Reference Application

The microservices reference application *T2-Project* is used as the basis for building and deploying the two variants. It simulates a web shop selling tea. The T2-Project was created at the *Institute of Software Engineering* at the University of Stuttgart with the aim of implementing and analyzing the *saga pattern* in a microservices application [SSB22]. It consists of seven microservices, all developed with *Java* and the *Spring Framework*[1] in conjunction with *Spring Boot*.[2] Apache[3] is used for asynchronous communication to handle the saga operations. The communication directly between the services takes place using Representational State Transfer (REST) interfaces synchronously. MongoDB[4] and PostgreSQL[5] are used as databases. The source code[6] and a documentation[7] of the T2-Project are publicly available. It is licensed under Apache 2.0[8] and can therefore be used as a basis for own research.

Originally the T2-Project is based on the ideas of the *TeaStore*[9] reference application, developed by the *Software Engineering Group* at the University of Würzburg with the purpose of microservices benchmarking [VES+18]. The TeaStore application consists of six microservices, five services and one central registration service. All services are developed with *Java Enterprise Edition (EE)* (today known as *Jakarta EE*[10]) and are deployed as *war* files with the webserver *Apache Tomcat*.[11] The T2-Project is used as the basis for this thesis instead of the TeaStore, because of the more modern and lightweight tech stack using Spring Boot. Interestingly, as shown by Calero et al. [CPM21], the usage of the Spring Framework can generally increase the productivity of developers significantly, but can also lead to an increase in energy consumption due to the higher level of abstraction and

[1] https://spring.io
[2] https://spring.io/projects/spring-boot
[3] https://kafka.apache.org
[4] https://www.mongodb.com
[5] https://www.postgresql.org
[6] https://github.com/t2-project/t2-project
[7] https://t2-documentation.readthedocs.io
[8] https://www.apache.org/licenses/LICENSE-2.0.html
[9] https://github.com/DescartesResearch/TeaStore
[10] https://jakarta.ee
[11] https://tomcat.apache.org

the heavy use of the reflection technique during runtime. However, the effects of the framework on energy efficiency are not discussed in this thesis. Both variants of the reference system are implemented with Spring Boot in order to keep the potential influences of the selected technologies within limits.

The microservices variant used for the study largely matches the original T2-Project with slight adjustments. The modular monolithic variant was newly created based on the source code of the individual microservices from the T2-Project and published in its own repository[12] as part of the same GitHub organization. In order to distinguish the individual variants and the overall project linguistically, the following terms are introduced:

- *T2-Project*: overall project, which includes both variants
- *T2-Microservices*: microservices variant
- *T2-Modulith*: monolith variant, retaining the modular structure

1.4 Cooperation With an Industry Partner

This thesis is written in cooperation with a partner from the IT industry: *envite consulting GmbH*.[13] envite is an IT consulting company based in Stuttgart, specializing in sustainable IT applications. The cooperation makes it possible in particular to derive recommendations for action from the results, to discuss their practical relevance and to incorporate them directly into the work of the consulting company.

1.5 Thesis Structure

This thesis is structured as follows:

Chapter 2 – Foundations: Here the foundations of the thesis are described.

Chapter 3 – Related Work: The related work to this thesis is presented here.

Chapter 4 – Reference Application: Describes the architecture of the T2-Project variants.

Chapter 5 – Measurement Methodology: Defines the measurement methodology.

Chapter 6 – Experiment Setup & Execution: Describes how the experiment was done.

Chapter 7 – Results & Discussion: This is where the results are presented and discussed.

Chapter 8 – Conclusion & Future Work Finally, a conclusion for this thesis is drawn.

[12] https://github.com/t2-project/modulith
[13] https://envite.de

2 Foundations

This chapter provides an overview of the fundamental principles and research that form the basis of this thesis. This includes a classification of the topic in the field of *green software engineering*, an introduction to *cloud computing*, a discussion of the *software architecture paradigms* relevant to this thesis, a list of *architecture recommendations by cloud providers*, an overview of the most important concepts of *energy consumption and GHG emissions of ICT*, how the *energy efficiency of software* can be assessed and an introduction to *green architectural tactics*.

2.1 Green Software Engineering

This work is part of the field of *Green Software Engineering*. According to Naumann et al. [NDKJ11], *Green and Sustainable Software Engineering* is characterized as the art of defining and developing software products in such a manner that the software product's overall life cycle is constantly assessed, documented, and used to enhance the software product, considering both the negative and positive effects on sustainable development that arise or are anticipated to arise from it. This is a rather broad definition that refers to the *17 Sustainable Development Goals (SDGs)*[1] of the United Nations.

ICT and Software in particular can have both a positive and a negative impact on the SDGs in a variety of ways. First-order, second-order and third-order effects can be distinguished [NDKJ11]. First-order effects refer to the direct impacts, e.g. energy consumption caused by software. The reduction of negative first-order effects in the area of IT is also referred to as *Green IT*. Second- and third-order effects are downstream impacts. Second-order effects are those effects where software has an impact on the life cycle of other products or services. Third-order effects are systematic effects such as the rebound effect, which leads to initial savings being overcompensated by additional use. *Green by IT* encompasses both second- and third-order effects and aims to reduce the negative environmental impacts in other sectors with the help of IT.

Under what conditions and to what extent digital solutions have a positive or negative impact can hardly be answered seriously and figures must be viewed with caution, taking into account the respective methodology [RLQ23]. What is clear, however, is that urgent action must be taken to tackle the many negative effects of ICT on the environment.

In order to achieve the sustainability goals in the ICT sector, it is not enough to focus solely on increasing energy efficiency [HA15]. Instead, all use of natural resources must be considered, not just energy, and negative second- and third-order effects must be taken into account. For example, the rebound effect can quickly lead to efficiency increases being completely offset by additional use or even to an increase in consumption.

[1] https://sdgs.un.org/goals

The question arises as to how software can be assessed for sustainability. Appropriate procedures and recommendations have been developed by Kern et al. [KHG+18] as part of the development of criteria for the certification of resource and energy efficient software products with the *Blue Angel*[2] environmental label of the German Environment Agency. Three aspects are relevant for the Blue Angel certification process: resource and energy efficiency, potential hardware operating life and user autonomy. However, the award criteria have so far only been defined for desktop application software that have a user interface. Criteria for server applications have been announced but have not yet been published at the time of writing this thesis.

This thesis focuses purely on the aspect of energy efficiency, without taking second- and third-order effects into account, in the hope that the findings of this thesis will not lead to any rebound effects.

2.2 Cloud Computing

This section defines what cloud computing and cloud-native mean and what the shared responsibility model is.

2.2.1 Definition

Cloud computing is a model that enables convenient and on-demand access to shared resources. The National Institute of Standards and Technology (NIST) defines cloud computing using five essential characteristics [MG11]:

- *On-Demand Self-Service:* Users can provision and manage computing resources as needed, without requiring human intervention from service providers.
- *Broad Network Access:* Services are accessible over the network and can be accessed through standard mechanisms, promoting use by various devices.
- *Resource Pooling:* Computing resources are pooled to serve multiple consumers, with different physical and virtual resources dynamically assigned and reassigned according to demand.
- *Rapid Elasticity:* Resources can be rapidly and elastically provisioned and released to scale with demand. This allows for quick scaling up or down based on application requirements.
- *Measured Service:* Cloud systems automatically control and optimize resource usage, providing transparency for both the provider and the consumer through metering capabilities. Users pay for only the resources they consume.

Cloud providers that offer a large number of servers and storage systems to the public via high-performance networks are also known as *hyperscalers* [EL21]. The term hyperscaler is derived from hyperscale computing, which is synonymous with massive scalability of computing resources. A hyperscaler provides IT resources based on the aforementioned cloud computing characteristics that are highly scalable horizontally, so there are virtually no upper limits to scalability.

[2] https://www.blauer-engel.de/en/productworld/resources-and-energy-efficient-software-products

As of October 2023, the three largest hyperscalers worldwide are Amazon Web Services (AWS) with a 32–34 % market share, Microsoft Azure with 23 % and Google with 11 % [Syn23]. Alibaba Cloud and IBM Cloud follow in fourth and fifth place with a market share of less than 5 % each.

The term *cloud-native* is used for applications that are operated *in the cloud* and utilize the advantages of cloud computing [GBS17]. According to Gannon et al. cloud-native applications fulfill at least five basic properties:

1. Operate at global scale: data and services are replicated globally
2. Scaling well with thousands of concurrent users
3. Resilient to failures
4. Continuous operation: Upgrading does not interrupt normal operations
5. Secure: Security must be part of the application architecture

The first design patterns for cloud-native applications emerged in 2013, which is also when the microservices architecture style became popular. Microservices are characterized by the fact that each microservice "must be managed, replicated, scaled, upgraded, and deployed independently of other microservices" [GBS17]. With *serverless computing*, cloud computing added a new paradigm that makes it possible to leave the execution of code completely to the cloud, in which only the code is written and events are defined [GBS17].

2.2.2 Shared Responsibility Model

When operating applications in the cloud, there is a twofold responsibility in terms of environmental sustainability: The cloud provider should ensure the most environmentally friendly cloud environment possible, i.e. environmentally friendly and efficient use of resources such as electricity, water, building materials, IT hardware, etc., and the customer should ensure the most environmentally friendly use of resources within the cloud, i.e. pay attention to code efficiency, utilization and scaling, software application design, etc. [Ama23; Mic23b]. AWS and Azure call this the *Shared Responsibility Model*.

This thesis deals exclusively with the responsibility and the possibilities on the customer side and will therefore not go into more detail about the sustainability efforts of cloud providers.

2.3 Software Architecture Paradigms

The architecture of a software system determines its fundamental characteristics and thus plays an important role in satisfying non-functional quality requirements. This section covers many foundational architectural paradigms: software complexity, domain-driven design, modularity, architectural styles, architectural tactics and data consistency in distributed systems.

The book “Domain-Driven Transformation” by Carola Lilienthal and Henning Schwentner from 2023 [LS23] is one of the sources used below. The book deals with how legacy software systems can be made future-proof with the help of domain-driven design and examines both monolithic and microservices architectures. In terms of content and due to its timeliness, it is very well suited for discussing the software architecture concepts that are fundamental to this thesis.

2.3.1 Software Complexity

Complexity in software development is a fundamental problem. Without countermeasures, the complexity of software inevitably increases over its lifetime, which leads to greater error rates, ever slower progress and poorer maintainability [LS23]. Complexity arises either from the domain (problem-inherent complexity) or due to the implementation during modeling, in the architecture or in the use of selected technologies (solution-dependent complexity). Two types of complexity can be distinguished: essential and accidental. Essential complexity is the type of complexity that is inherent in something, i.e. part of its essence (domain, technology, etc.). This complexity can never be resolved or avoided by a particularly good design. Accidental complexity, on the other hand, refers to complexity that could have been avoided, e.g. through better communication within the team, a more suitable architecture for the problem or a more sensible selection and use of technologies.

Good software development avoids accidental complexity and makes essential complexity manageable. The following section discusses methods that can help with this.

2.3.2 Architecture Documentation

To recognize complexity, it helps to make it visible with the help of architecture documentation [LS23]. There are many different approaches for this. In this thesis, *Unified Modeling Language (UML)*,[3] the *C4 model*[4] by Simon Brown and *arc42*[5] by Gernot Starke and Peter Hruschka are used. These have their own strengths and can complement each other well.

arc42 is way of creating a complete software architecture documentation in a pragmatic way. It is licensed under the CC BY-SA 4.0 license.[6] The C4 model is a lightweight variant for the visualization of architecture. The associated website and the examples are available under the CC BY 4.0 license.[7] The C4 model is based on approaching from the coarse to the fine and considering four levels: Context, Container, Components, Code (hence the name C4 model). Context describes the overall system and its stakeholders, containers divide the software into executable parts, components define the internal implementation details (several components run in the same process), and code shows the concrete implementations of the components. UML can also be used to visualize architecture. UML is internationally standardized and has been around for about three decades. Compared to the

[3] https://www.uml.org
[4] https://c4model.com
[5] https://arc42.org
[6] https://arc42.org/license
[7] https://creativecommons.org/licenses/by/4.0

C4 model, it is significantly more comprehensive, but also less intuitive. It still has its advantages mainly due to its standardization, familiarity and prevalence. In this thesis, UML is used in the form of sequence diagrams for the visualization of dynamic behavior.

It is advisable to create diagrams as code in order to generate both the documentation and the software automatically and to be able to reuse components more easily as well as to enable version control. For this thesis, the tool PlantUML[8] is used, which supports both UML and C4 using the library C4-PlantUML.[9] PlantUML and C4-PlantUML are both licensed under the MIT license and can be used free of charge.

2.3.3 Domain-Driven Design

Domain-Driven Design (DDD) is a conceptual approach to the development of complex software, with the aim of making the complexity of the business domain understandable, explicitly integrating this knowledge into the design of the software and avoiding accidental complexity [LS23]. DDD was originally developed by Eric Evans, who published "Domain-Driven Design" in a book of the same name in 2004. Today, his book is best known as the *Blue Book*. In 2013, Vaughn Vernon published the book "Implementing Domain-Driven Design", which is now known as the *Red Book* and has expanded DDD with further concepts. Fundamental to domain-driven design is the focus on technicality, not only in requirements elicitation, but especially in modeling and programming. Cooperation between domain experts and software developers is essential for modeling. To avoid misunderstandings, DDD proposes the use of a common, universal language between developers and domain experts: the *ubiquitous language*. Another fundamental concept is *bounded contexts*. Bounded contexts in DDD define delimited areas within an application domain in which certain terms and rules have a clear meaning. They help to structure complex domain models by setting clearly defined context boundaries. Each bounded context can have its own model and language, which helps to avoid misunderstandings and improve communication between developers and subject matter experts.

The topic of domain-driven design as a whole is very broad, so that only the aspect of modularity will be discussed further here.

2.3.4 Modularity

Modularity is an important component of DDD and essential for making complexity manageable. For this purpose, the large complex problem is divided into several small manageable units and provided with narrow interfaces that make the functionality accessible to others (*information hiding*) [LS23]. A modular architecture also has the advantage that the implementation of individual modules can be easily changed as long as the interface does not change. The concept of modularity was first introduced by David Parnas in 1972. It is therefore a comparatively old concept, but one that plays a very important role in software engineering.

[8] https://plantuml.com

[9] https://github.com/plantuml-stdlib/C4-PlantUML

Modules with high internal concentration and low interaction with other modules are described with the characteristics *high cohesion* and *less coupling* [LS23]. Cohesion measures the degree of closeness within modules, while coupling describes the degree of dependency between the modules of a software system. A higher number of dependencies leads to a stronger coupling in the system. If the software system has an interwoven structure with high coupling and low cohesion, the system is commonly referred to as a *Big Ball of Mud* [LS23]. This term was originally coined by Foote & Yoder in 1997 [FY97]. A Big Ball of Mud arises automatically over the time of an ongoing software development if no refactorings and no architecture renewal are carried out. Regular refactorings to tidy up the *mess* are therefore essential for long-term maintainable software. Here it is particularly helpful to modularize the software in a sensible way in order to equalize the interwoven structures and only allow components to communicate via narrow, well-defined interfaces.

However, modularizing a software system in a meaningful way is no trivial task. This is where domain-driven design comes into play. The concept of bounded contexts can be used as a good basis for the organization of modules. Once bounded contexts have been defined using DDD, an independent module can be implemented for each bounded context.

Eric Evans describes modularization as follows [Eva15]:

> Choose modules that tell the story of the system and contain a cohesive set of concepts. Give the modules names that become part of the ubiquitous language. Modules are part of the model and their names should reflect insight into the domain.

2.3.5 Modular Architecture Styles

A modular software architecture can be implemented with both a monolithic and a distributed system. A modular monolith has an internal modular structure, but is deployed as a single unit. Microservices is an architectural style for a modular distributed system. Whether monolithic or distributed, modularity should always be a fundamental goal in software architecture work, as already discussed. A big ball of mud can arise not only in a monolithic system, but also in a microservices system, as shown in Figure 2.1, created by Simon Brown [Bro18].

Microservices

About 10 years ago, the microservices architecture style began to become popular as a way to develop complex cloud applications with large teams and operate them at scale [LF14]. Netflix is often cited as a prominent example of a large microservices system that has helped make the architectural style so popular. The microservices architectural style is based on the principle that an application consists of multiple independent and loosely coupled services, each responsible for a specific business function [LF14]. Each microservice can be written in its own programming language and deployed independently. This division allows teams to work on different services simultaneously, increasing scalability and development speed. Communication between microservices is done through standardized Application Programming Interfaces (APIs).

Another characteristic of microservices architectures is the way in which data is stored. To ensure the loose coupling of the services, it is common in a microservices architecture to assign microservices their own databases [Ric17a]. This pattern is called *database per service*. It also has the advantage

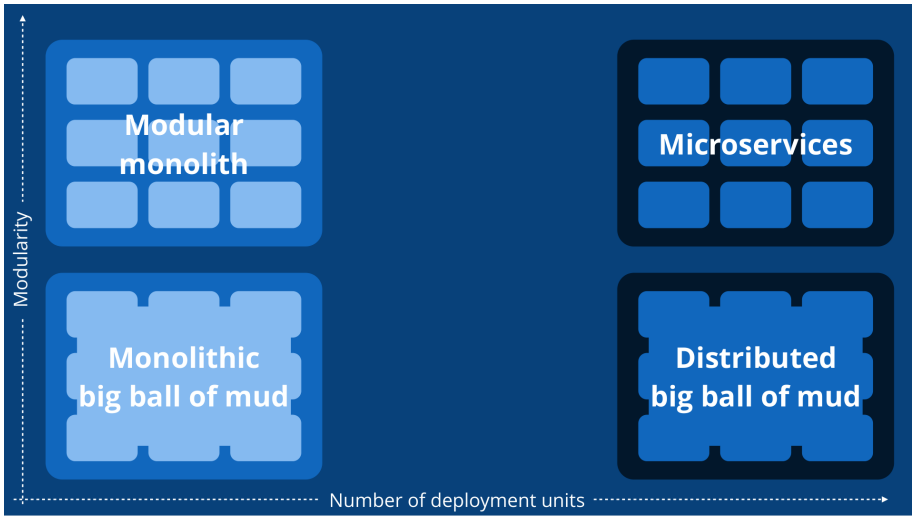

Figure 2.1: Modular architecture styles vs. big ball of mud. Source: [Bro18]

that database types can be selected for each microservice that best suit the respective requirements. The disadvantages of the pattern are that transactions become more complicated, the linking of data sets from different services becomes more difficult and the complexity of operation increases due to the management of several, possibly different databases.

Overall, the microservices architecture style is associated with greater complexity due to the distributed system, which can have a negative impact on comprehension, maintainability, testability and operation. The implementation of transactional workflows is much more difficult to implement with microservices, which is why various approaches are used in practice to deal with this. The Section 2.3.7 “Transactions in Distributed Systems” is dedicated to this topic.

Modular Monolith

A search on Google Scholar for “Modular Monolith” reveals that, on the one hand, this term is still used comparatively little and, on the other hand, several publications have appeared with this term since 2023. A recent paper poses the appropriate question right in the title: “Modular Monolith: Is This the Trend in Software Architecture?” [SL24]. Su and Li used a systematic literature review to look at what constitutes the architectural style and what approaches to implementation already exist in practice. They conclude that a modular monolith combines the simplicity of the monolith with the advantages of microservices and, in particular, makes it easy to separate and deploy individual modules later as microservices. It is also emphasized that the focus on business domains rather than technical levels is important to ensure good code organization and maintainability.

Su and Li highlight four cases in which the architectural style of a modular monolith is used. Of particular note here is Shopify,[10] one of the largest applications with a Ruby on Rails codebase, on which more than a thousand developers have already worked. Shopify has explicitly decided against the microservices architecture style in order to avoid the associated disadvantages, such

[10] https://www.shopify.com

as infrastructure overhead, latencies and decrease in reliability due to network communication and difficulties of refactoring across service boundaries [Wes19]. To still create a clear separation between component boundaries, Shopify decided to evolve the original monolithic architecture into a modular monolith. As part of the transition, Shopify developed a tool internally called Wedge, which is used to monitor the isolation of components and ensure the decoupling of business domains.

According to the paper by Su and Li, there are currently three frameworks for creating modular monoliths: *Service Weaver*, *Light-hyrid-4j* and *Spring Modulith*. These three are presented next.

Service Weaver is an exciting new approach from Google and has received a lot of attention in 2023 with the publication "Towards Modern Development of Cloud Applications" [GGP+23]. The authors describe the new approach as follows: "developers write their applications as logical monoliths, offload the decisions of how to distribute and run applications to an automated runtime, and deploy applications atomically" [GGP+23]. The programming framework is available as open source under the name Service Weaver,[11] but is still in an early development phase (version 0.22) at the time of writing this thesis. The authors generally advocate that developers should implement modular monoliths and only later decide whether it is really necessary to switch to a microservices architecture. The runtime-based approach presented promises the advantages of microservices, such as scalability, but with significantly better performance, reduced costs and less complexity during development and operation. The extent to which this new approach will become widespread in the industry remains to be seen. It is also unclear what effect this approach will have on energy efficiency. However, the improved performance and the possibility of scalability promise that Service Weaver could potentially also have a positive impact on energy efficiency. Due to the still young stage of development and the limitation to Go applications, this new approach does not play a role in this thesis. However, the publication and the response it has received illustrate the current upswing in the aspect of modularity in monoliths.

The light-hybrid-4j[12] framework takes a different approach and uses jar files to separate individual services or modules. In contrast to microservices, however, several jar files are provided in a shared Docker container volume[13] and executed within the same Java Virtual Machine (JVM). Here too, the framework promises that efficient scaling is possible if required by separating the relevant parts into separate containers. light-hybrid-4j is designed for use with the Light platform,[14] a serverless platform for cloud-native applications. For this reason, this framework is not suitable for this thesis.

Spring Modulith[15] is a new Spring project that aims to help organize source code based on the module concept. It comes as a library for Spring Boot applications and is able to validate the structure, document the module design, perform integration testing of individual modules, observe module interaction at runtime, and implement module interaction in a loosely coupled manner. Spring Modulith is still a very new project. Version 1.0 was only released shortly before the start of this thesis. However, it has already received a lot of attention and is currently listed in the

[11] https://github.com/ServiceWeaver/weaver
[12] https://github.com/networknt/light-hybrid-4j
[13] https://docs.docker.com/storage/volumes
[14] https://doc.networknt.com
[15] https://spring.io/projects/spring-modulith

Assess ring of ThoughtWorks' *Technology Radar*,[16] i.e. it is considered interesting and it is worth keeping an eye on the project. Spring Modulith is a perfect match for this thesis. For the task of creating a modular monolith from existing Spring Boot microservices, Spring Modulith provides best practices and helps to form an actual modular monolith.

2.3.6 Architectural Tactics

Architectural styles and patterns typically encompass several design decisions and affect various quality attributes accordingly. For example, the microservices architectural style deals in particular with the six quality aspects of scalability, performance, availability, monitorability, security, and testability [LZJ+21]. Architectural tactics are simpler than patterns and aim to achieve a single desired quality attribute response [BCK21]. For example, two tactics are discussed in research for the quality goal of horizontal scalability, one of the most important properties of microservices architectures: horizontal duplication and vertical decomposition [LZJ+21]. Horizontal duplication refers to replicating identical parts of an application on multiple nodes or instances, while vertical decomposition is the division of an application into different layers or services to support specific functionalities, including enabling a clear separation between data management and business logic.

Strategic trade-off decisions must be made when applying tactics. For example, when applying the tactic of vertical decomposition, the optimal granularity of microservices must be determined to balance scalability and performance, taking into account factors such as high cohesion and loose coupling [LZJ+21]. The challenge is to avoid overly small microservices, which may increase scalability but may also negatively impact performance due to increased inter-service communication.

Tactics that aim to have a positive impact on environmental sustainability are known as *Green Architectural Tactics*. Most of these are about improving energy or carbon efficiency. This thesis here focuses on the tactics that deal with the quality goal of energy efficiency.

2.3.7 Transactions in Distributed Systems

In monolithic architectures, a transaction with Atomicity, Consistency, Isolation, Durability (ACID) guarantees can be executed across module boundaries. In the microservices architecture style, on the other hand, the execution of a cross-service transaction is not easily possible due to the distributed nature of the system. However, there are various architectural tactics for enabling transactions with certain trade-offs in distributed systems. Classic distributed transactions, usually implemented with the two-phase commit protocol, are rarely used in practice with microservices architectures due to feared performance losses [FPG+23; LZS+21]. Instead, in practice, microservices usually rely on eventual consistency as a consistency model, which promises better performance and scalability, but comes at the expense of strict consistency. According to Laigner et al. and Ferreira Loff et al., orchestration-based mechanisms are often used in practice for the implementation of cross-service workflows, including the saga pattern. The saga pattern enables the use of transactions

[16] https://www.thoughtworks.com/en-de/radar/languages-and-frameworks/spring-modulith

with distributed services by splitting transactions into smaller steps and performing compensating actions when errors occur [Ric17b]. Therefore it is often also categorized as a failure management pattern.

Accordingly, sagas can only be used in applications where compensation mechanisms are available. One disadvantage of using the saga pattern is that the complexity of the implementation is significantly increased. It should also be noted that either *event sourcing* or the *transactional outbox pattern* must be used for reliable execution of sagas [Ric17b]. For this thesis, only the transactional outbox pattern will be relevant, which is why event sourcing will not be discussed here. The transactional outbox pattern solves the problem that writing to a database and sending a message is not possible atomically within a transaction [Ric17c]. The pattern introduces an additional outbox table in the database to store messages about successful transactions. An external service periodically queries this outbox table and sends new entries to other services. As polling or similar methods are used here, there are delays in the processing of the workflow.

In order to make the complexity of the saga pattern manageable, a framework should be used for implementation. One of these frameworks is *Eventuate Tram*,[17] which is comparatively easy to use and works well with Java and Spring [DLW22; ŠCR19]. Štefanko et al. examined, among other things, how different saga frameworks perform in terms of performance. In the test scenario with 1000 requests and 10 threads, the processing delay, the time between the last request and the end of the last transaction, was between 4 and 49 seconds, depending on the framework. The Eventuate Tram service, which will be relevant for this thesis, had a delay of 27 seconds in this test.

The major cloud operators provide their own recommendations on how they believe transactions should be implemented in cloud applications. These recommendations are discussed in the next section.

2.4 Architecture Recommendations by Cloud Providers

Each of the cloud providers provides their own recommendations on how they believe a cloud-native application should be designed, developed, tested and operated and how various challenges should be dealt with. The architecture recommendations of the three major cloud providers AWS, Azure and Google with regard to environmental sustainability and the implementation of transactional workflows are particularly relevant for this thesis, so these are discussed below. The aspect of transactional workflows is dealt with explicitly here, as transactions represent a particular challenge in the implementation of distributed systems and therefore may have a potentially major impact on the application and therefore also on energy consumption. The recommendations are summarized at AWS, Azure, Alibaba and IBM under the name *Well-Architected Framework*[18,19,20,21] and at Google under the name *Cloud Architecture Framework*.[22]

[17] https://eventuate.io/abouteventuatetram.html
[18] https://aws.amazon.com/architecture/well-architected/
[19] https://learn.microsoft.com/en-us/azure/well-architected/
[20] https://www.alibabacloud.com/architecture
[21] https://www.ibm.com/architectures/well-architected
[22] https://cloud.google.com/architecture/framework

In the following, only the architecture recommendations of the three largest cloud providers, AWS, Azure and Google, will be discussed. Alibaba Cloud doesn't provide any sustainability recommendations yet, and IBM's recommendations are quite brief and don't provide any additional information worth discussing here.

2.4.1 Sustainability Recommendations

The aspect of environmental sustainability is anchored differently in the architecture recommendations of the three cloud providers: In the AWS Well-Architected Framework, sustainability is one of six fundamental pillars, called *Sustainability Pillar* [Ama23], at Azure sustainability is presented as workload guidance, called *Sustainability Workload* [Mic23b], and at Google, sustainability is mainly discussed in one article with the title "Reduce your Google Cloud carbon footprint" as part of the *system design* pillar in their Cloud Architecture Framework [Goo21]. The framework contains another sustainability related article called "Designing for environmental sustainability",[23] however, it is quite short and mainly refers to the aforementioned article on reducing the carbon footprint of the cloud, so it doesn't provide any additional relevant information.

There are, of course, many recommendations that can lead to more sustainable use of the cloud during design, development, testing and operation. The following are just a few of the cloud provider recommendations for environmental sustainability that are relevant to this thesis.

The sustainability pillar of the AWS Well-Architected Framework does not provide many specific recommendations for action with regard to the most sustainable software architecture possible, but it does mention a variety of points that software architects should think about [Ama23]. These include, in particular, the question of trade-offs, as some quality aspects could lead to a negative impact on environmental sustainability: High availability, performance optimization by adding more resources or a high security level However, it is also mentioned elsewhere that trade-offs in other quality aspects do not have to be made to the same extent in order to achieve the sustainability goals. Even small trade-offs can lead to significant improvements in terms of sustainability. It is emphasized several times that maximizing the utilization of the resources used is very important in order to improve energy efficiency. As an example, it is stated that two machines with a 30 % utilization rate are less efficient than one machine with a 60 % utilization rate. Accordingly, it is recommended to continuously monitor the utilization of the resources used and to remove or refactor resources with a low utilization. Furthermore, it is recommended to use managed cloud services, as shared services increase utilization and the highly optimized managed services should generally contribute to an improvement in energy efficiency. With regard to the architecture design, it is particularly recommended to rely on asynchronous communication where possible and to use queues in order to ensure even scaling of the components. With regard to communication, it is generally recommended to minimize data transfer as far as possible. If changes are to be made, it is recommended that suitable proxy metrics are evaluated at the same time in order to quantify the impact of the changes. Examples include the duration of the utilization of a virtual Central Processing Unit (vCPU) for a compute-relevant change, the required storage space for a storage-relevant change and the amount of data transferred for a network-relevant change.

[23] https://cloud.google.com/architecture/framework/system-design/sustainability

Microsoft Azure has created the sustainability workload documentation in cooperation with the Green Software Foundation so that each recommendation is aligned with the *principles of green software*[24] defined by the Green Software Foundation (GSF) [Mic23b]. With regard to the architectural style, it the sustainability workload documentation specifically recommended to evaluate the move from monoliths to microservices. The scalability of individual components during peak load is emphasized as an advantage, which also allows the scaling down or in of idle components. However, trade-offs of the microservices architecture style are also mentioned, which should be taken into account: network traffic could increase, the complexity of the application could increase significantly and there could be an overhead in deployment. In terms of measurement, it is recommended to use cost as a proxy metric, as it is "one of the safest and most potent proxies for carbon emissions" [Mic23b].

Google also recommends refactoring the software architecture from a monolithic style to a microservices style [Goo21]. As all modules in a monolith are located in one program, operating and scaling monolithic applications efficiently can be more difficult. This could result in a higher carbon footprint compared to a microservices-based implementation. One example given is a store where users interact much more frequently with the shopping cart service than with the payment service. If the two services are implemented as microservices, they can be scaled individually and independently of each other, whereas with a monolith, the entire application always has to be scaled, which would lead to increased resource consumption. However, in the whitepaper "Go Green Software Guide", Google also highlights the side effect of the microservices architecture style that more communication takes place over the network [Clo24]. Google points out that a 'chatty' microservice can be more wasteful than a monolith. Google therefore explicitly recommends that architects should consider how large the transmitted payload is, how great the distance between the services is and whether the appropriate transmission protocol is used.

Finally, a brief summary: None of the cloud providers make recommendations on how energy consumption or carbon emissions can be recorded at development time. Instead, they refer to proxy metrics that should be used to evaluate changes in terms of sustainability. Their carbon dashboards, discussed in the Section 2.5.5 "GHG Reporting in the Cloud", are primarily for reporting purposes, not really helpful for usage in everyday development and maintenance. In terms of architecture design recommendations, it can be said that all cloud providers emphasize the advantages of microservices or in general of fine-grained components.

2.4.2 Data Consistency Recommendations

AWS does not address the aspect of data consistency in cloud-native applications as part of their Well-Architected Framework, but in a guide as part of the *AWS Prescriptive Guidance*[25] entitled "Enabling data persistence in microservices" [WM23]. In this guide, the saga pattern is mentioned as a way to establish consistency in distributed systems. It is emphasized that it is difficult to debug, the complexity increases with the number of microservices and it requires a complex programming model. The implementation and trade-offs of the saga pattern and the transactional outbox pattern

[24] https://learn.greensoftware.foundation/introduction
[25] https://aws.amazon.com/prescriptive-guidance/

are described in more detail in the AWS Prescriptive Guidance "Cloud design patterns, architectures, and implementations" [Dee23]. The patterns are illustrated with the help of serverless demo applications.

Azure explicitly recommends the use of the patterns *saga*[26] and *transactional outbox*[27] for applications with high reliability [Mic23a] . The various advantages and disadvantages of the patterns and the different variants of the saga pattern are explained in detail. No recommendation for a specific implementation is given. However, an example application called "Orchestration-based saga on Serverless"[28] is provided that shows how to implement an orchestration-based saga with the serverless paradigm.

Google recommends the use of the saga pattern in its tutorial "Interservice communication in a microservices setup" as part of the *Google Cloud Architecture Framework*, especially for long-lived transactions [Goo23]. Reference is made to several workflow engines that can help with the implementation of sagas. However, further information on implementation is not provided in this article and the saga pattern is not implemented in the referenced microservices demo application *Online Boutique*,[29] so that data consistency is not guaranteed during an order process in this demo shop. In another tutorial article entitled "Transactional workflows in a microservices architecture on google cloud", the implementation of a transactional workflow is discussed in more detail, in which the two patterns *choreography-based saga* and *synchronous orchestration* are discussed [Goo22]. However, the compensation mechanisms required in practice as part of a saga are missing.

To summarize, all of the cloud providers listed here recommend the saga pattern and the transactional outbox pattern if transactional integrity over multiple services is to be maintained. AWS and Azure are very transparent about the challenges and disadvantages of applying the patterns, which is not clear from Google's documentation.

2.5 Energy Consumption and GHG Footprint of ICT

This section presents the basics of energy consumption as well as the GHG footprint in the context of ICT. As explained in the Introduction chapter, the thesis focuses on the energy efficiency of software and not explicitly on the reduction of GHG emissions. Nevertheless, understanding the calculation of a GHG footprint is helpful to better understand the importance of improving the energy efficiency of software.

2.5.1 Metrics

There are a few metrics and units that are relevant for energy and climate related measurements and improvements. The following explanations are inspired by the article "All you need to know about Energy Metrics in Software Engineering" by Luís Cruz and Philippe de Bekker [CdB23].

[26] https://learn.microsoft.com/en-us/azure/architecture/reference-architectures/saga/saga
[27] https://learn.microsoft.com/en-us/azure/architecture/databases/guide/transactional-outbox-cosmos
[28] https://github.com/Azure-Samples/saga-orchestration-serverless
[29] https://github.com/GoogleCloudPlatform/microservices-demo

Energy is a fundamental concept in physics, representing the ability to do work. The standard unit of energy is *Joule* (J), where 1 Joule is equal to the energy transferred when a newton of force is applied to an object in the direction of the force over a distance of one meter. In the context of electrical energy, the joule is mainly used in scientific literature, while in everyday life it is more common to use the unit *kilowatt-hour* (kWh). The kilowatt-hour serves as a practical unit for measuring the amount of electricity consumed over time. Kilowatt-hours can be easily converted to joules and vice versa as follows:

$1\ kWh = 3,600,000\ J$

Power, on the other hand, is the rate at which energy is transferred. The unit of power is *watt* (W), where 1 watt is equal to 1 joule per second.

According to Cruz and de Bekker, the metric of energy consumption is usually more important than power consumption when evaluating the energy efficiency of software. Typically, the goal is to capture the energy consumption for a specific use case and not just the power consumption at a specific point in time. Cruz and de Bekker give the examples of applying a filter to a photo and uploading a file to a server. In order to be able to make a statement about energy efficiency, the entire duration of the execution must be taken into account. Measuring the power consumption alone only makes sense if it is relatively constant and the duration does not play a decisive role. Reading an e-book is given as an example.

The energy consumption (W) can be calculated as follows, assuming a known average power consumption (P_{avg}) and duration of operation (Δt):

$W = P_{avg} \times \Delta t$

However, the power consumption of many software applications is not normally constant, as the Figure 2.2 symbolically illustrates (source: [CdB23]).

The energy consumption is the integral of the power consumption over the interval of time required for the execution of the application to be measured:

$W = \int_{t_0}^{t_n} P(t)dt$

The climate impact of energy consumption is expressed as carbon intensity. Carbon intensity is the amount of carbon dioxide equivalent emitted per unit of energy consumed and is typically expressed in $\frac{g\ CO_2eq}{kWh}$. Carbon intensity can vary greatly due to various external factors. For example, the amount of emissions varies considerably depending on the time of day and location due to the type of electricity generation (wind, solar).

2.5.2 Energy Consumption in Hyperscale Data Centers

As this thesis focuses on the energy efficiency of cloud-based software, this section discusses how the energy consumption in the hyperscalers' data centers comes about and what optimization potential there is.

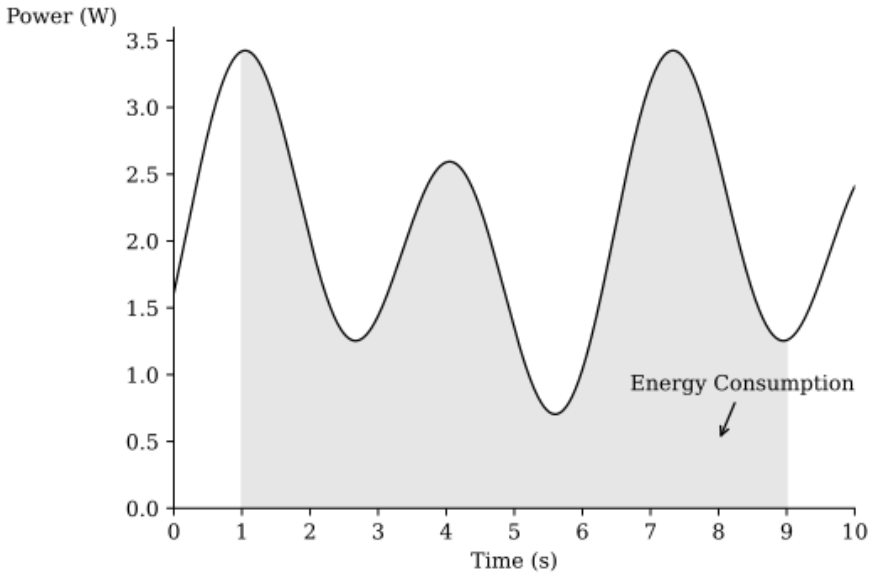

Figure 2.2: Plot of power consumption over time. The area in gray is equivalent to the energy consumption between t=1s and t=9s. Source: [CdB23]

Efficiency of Data Centers

For cloud computing, it is fundamental that a large pool of hardware resources is available at all times. This requires large data centers, where there is a great incentive for operators to run them as efficiently as possible and thus save costs. A popular metric for the efficiency of a data center is *Power Usage Effectiveness (PUE)*. The metric indicates the ratio between the total power required by the data center and the power consumed by the IT equipment (e.g. servers, network and other IT equipment) [BHR19]. A PUE of 1.5 would mean, for example, that two thirds of the total power consumption of a data center is used for IT components and one third for other data center equipment such as cooling and power distribution losses. Today's hyperscaler data centers have a PUE of less than 1.2 and thus perform very well compared to on-premise data centers in terms of energy efficiency [BHR19]. However, the metric also has some weaknesses, meaning that focusing solely on it is not appropriate from a sustainability perspective. For example, the PUE provides no indication of how efficient the IT components are (in fact, the PUE tends to improve when energy-inefficient IT components are used).

On the one hand, this makes it clear that the optimization potential on the infrastructure side for non-IT components has been almost completely exhausted for hyperscalers and, on the other hand, that there is now an urgent need to focus on increasing the efficiency of IT components.

Efficiency of IT Components

The aspect of *energy proportionality* is a challenge for the efficiency of server components, which means that the power consumption is not proportional to the utilization. Due to their high static power, they consume a considerable amount of energy even when idle. However, a lot has happened

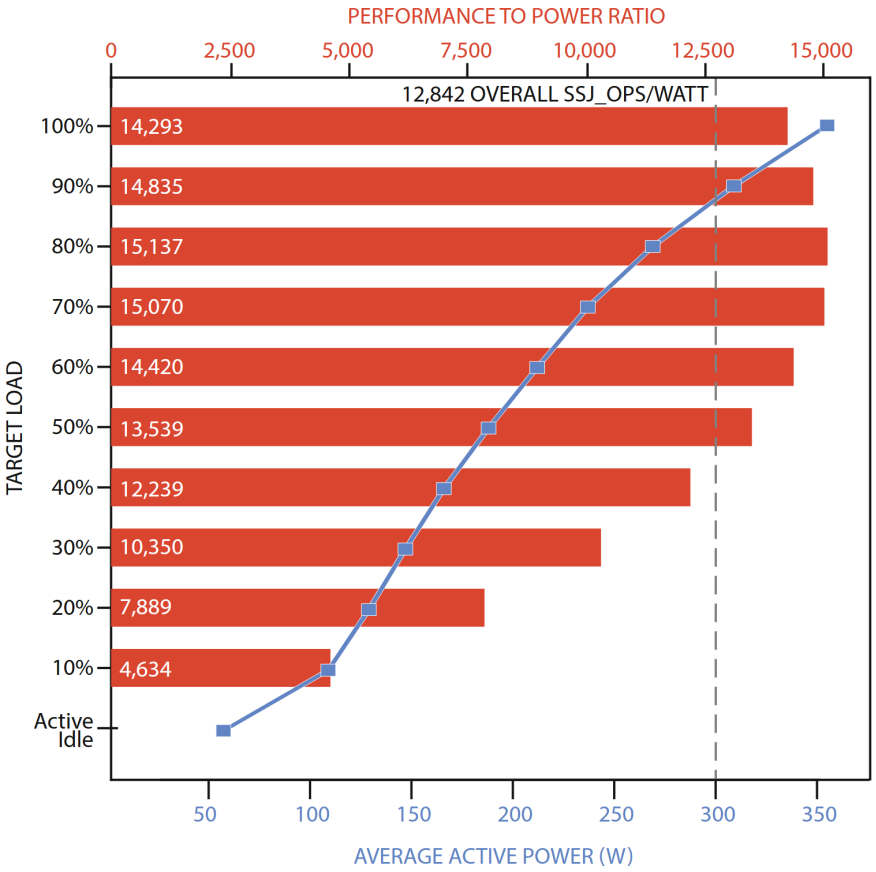

Figure 2.3: Benchmark result for SPECpower_ssj2008; bars indicate energy efficiency and the line indicates power consumption. Both are plotted for a range of utilization levels, with the average energy efficiency metric corresponding to the vertical dark line. Source: [BHR19]

in recent years. In 2007, a server in Google data centers already required over 60 % of the maximum power consumption in idle mode; in 2018, the value was less than 20 % [BHR19]. However, there is still no energy proportionality. Figure 2.3 illustrates this using a result from the benchmark *SPECpower_ssj2008*[30] from 2018 at different load levels. The red bars show the energy efficiency – ratio of transactions per second to power consumption – and the blue line shows the power consumption. The graph clearly illustrates that the efficiency of the system increases significantly with higher utilization and is at its highest in the 70–90 % utilization range. Even when the system is inactive, it still consumes just under 60 W, which in this case accounts for 16 % of the server's peak power consumption.

It is unclear how high the server utilization in the hyperscalers' data centers actually is. Luiz André Barroso, who was responsible for the redesign of Google's data centers and servers, assumes that typical server clusters in Google data centers have an average utilization of 10–50 % [BHR19]. Shehabi et al. also assumed in 2016 that the average server utilization for hyperscalers will be 50 %

[30] https://www.spec.org/power_ssj2008

in 2020, while it will be 25 % for service provider data centers and only 15 % for in-house servers [SSS+16]. More recent studies are not known. It can be concluded that the servers in the cloud are not fully utilized, but the utilization is at least significantly higher than is the case with typical on-premise servers.

In order to increase the utilization of underutilized resources, hyperscalers offer so-called *spot instances*.[31,32,33] They are offered at a lower price than on-demand instances, so there is a financial incentive to prefer them. However, spot instances can be revoked by the cloud operator at any time, meaning that they are only suitable for certain workloads and the software system must be able to deal with it.

The question of which of the IT components requires the most power is easy to answer: The Central Processing Unit (CPU) requires by far the most power, especially under load. Under full load, the power consumption of the CPU of a server in Google's data centers accounts for around two thirds, and 40 % in idle mode [BHR19]. However, this also shows that the energy proportionality of other IT components such as memory, storage and network components is an even bigger challenge. Network components in particular consume almost as much energy at idle as under full load (dynamic range for switches is 1.2) [BHR19].

2.5.3 Energy Consumption of Network Data Transfer

The total energy consumption of a cloud-based software system includes not only the energy consumption on the server side but also the energy consumption that occurs during data transfer between data centers and on the way to the users. The proportion of global energy consumption and global GHG emissions accounted for by data traffic on the internet, and how this proportion will develop in the coming years, is a question that has been the subject of heated debate in both popular science and scientific publications in recent years. The question usually revolves around what constant can be assumed for the average energy consumption when transferring one gigabyte of data on the internet.

According to Aslan et al. [AMKF18] the value for 2015 for data transfer in the Internet Protocol (IP) core network and fixed-line access (e.g. Digital Subscriber Line Access Multiplexer (DSLAM)) was 0.06 kWh/GB. The authors assume that this value halves every two years. Accordingly, the value for 2024 could have been reduced to 0.00265165 kWh/GB ($0.06 \times 0.5^{\frac{9}{2}}$). Of course, there is a great uncertainty in this estimate. However, this constant is the best we have to date. To arrive at this figure, Aslan et al. analyzed 14 studies and discussed the different methodologies used. This number is currently used by many to calculate the GHG emissions of data transfer over the Internet, such as the Carbon Trust's comprehensive study of video streaming emissions [STF+21] or the Green Metrics Tool [Tar24]. The Cloud Carbon Footprint tool (presented in Section 2.5.6) uses the constant 0.001 kWh/GB for data transfer between geographically decoupled data centers. Data transfer within a data center and data transfer to users is not included.

[31] https://aws.amazon.com/ec2/spot

[32] https://azure.microsoft.com/en-us/products/virtual-machines/spot

[33] https://cloud.google.com/spot-vms

However, the use of a constant for kWh/GB is generally criticized, as a discussion on the mailing list "e-impact"[34] of the Internet Engineering Task Force (IETF) illustrates: "kWh/GB should be banned and the IETF should warn against its use". This raises the question of the extent to which data transfer should even be included in the calculation of energy consumption or the carbon footprint of a software application.

As already mentioned, energy proportionality is a major problem with IT components and network devices in particular. The energy consumption of today's network devices is essentially independent of their load and comes almost exclusively from switching on the device [JV23]. A change in the data volume of a software application therefore does not necessarily have to have an impact on the energy consumption of network devices. What can be recorded and optimized at software level, however, is the computing time required to process the data, both on the server side and on the client side. A reduction in the volume of data or adjustments to the method of compression or encryption can therefore help to reduce the overall energy consumption of the software.

2.5.4 GHG Protocol

The *GHG Protocol*[35] is an internationally recognized standard for measuring and reporting greenhouse gas emissions, for entire companies or just for individual products. The emissions are divided into three *scopes* [Ins11]:

- *Scope 1* emissions represent the direct emissions that are "owned or controlled" by the reporting organization, such as the combustion of fuel in owned boilers.
- *Scope 2* emissions represent the indirect emissions from the generation of purchased energy, such as electricity, heating, cooling, etc.
- *Scope 3* extends the indirect emissions into the company's value chain and considers both upstream and downstream activities, such as the production of purchased products or the use of sold products. Upstream emissions occur before the company's own goods or services are produced, e.g. through the procurement or purchase of required goods and services, and downstream emissions are the indirect emissions caused by the sale of goods and services.

In 2017, an extension of the Product Standard[36] of the GHG Protocol was published specifically for the ICT sector: "ICT Sector Guidance built on the GHG Protocol Product Life Cycle Accounting and Reporting Standard" [TI17]. This guidance establishes a framework and provides helpful information and recommendations on how the accounting and reporting of GHG emissions from ICT products can and should be carried out. Software applications and software services are also considered as part of this guidance. In the following two sections, some recommendations from the ICT Sector Guidance are reproduced. Further sources are consulted in subsequent sections in order to examine individual aspects in more detail, to reproduce discussions from the scientific literature or to provide examples of practical applications.

[34] https://mailarchive.ietf.org/arch/msg/e-impact/t9lzF7j2DQWi4EDp7Pe2JKlcQdg
[35] https://ghgprotocol.org
[36] https://ghgprotocol.org/product-standard

Assessing GHG Emissions Related to ICT Products

The GHG footprint of an activity is specified in the unit kg CO_2eq or kg CO_2e and also includes other GHG emissions in addition to the greenhouse gas carbon dioxide (CO_2). When accounting for ICT services, only CO_2 emissions are often mentioned, as other GHG emissions play a subordinate role in the ICT sector in the vast majority of cases. Accordingly, this thesis will mostly refer to carbon emissions. In this foundational section, however, carbon dioxide equivalent (CO_2e) will be used as a unit in order to be consistent with the literature on the GHG Protocol.

The GHG footprint is generally calculated using a quantified measure of an activity multiplied by an emission factor:

GHG Impact ($kg\ CO_2e$) = Activity Data ($unit$) $\times$ Emission Factor ($\frac{kg\ CO_2e}{unit}$)

Such an activity data can be for instance the energy consumption of an ICT device. The energy consumption multiplied by the emission factor for electricity then gives the CO_2e footprint for use in the respective period. To illustrate this, an example scenario is given of a router that is operated without interruption and requires a constant 800 W. With an emission factor of 0.60 kg CO_2e per kWh, this gives the following result for the CO_2e footprint in one day:

GHG Impact = 19.2 ($kWh\ per\ day$) $\times$ 0.60($\frac{kg\ CO_2e}{kWh}$) = 11.5 ($kg\ CO_2e\ per\ day$)

To calculate the CO_2e footprint for use, both the exact power consumption and the emission factor should ideally be known. However, both are variable over time in practical applications. The electricity consumption changes depending on the utilization and the emission factor depending on the electricity mix in the grid of the particular region at the particular time (assumption location-based approach without own electricity generation). The GHG Protocol does not yet contain any strict statements on how carbon intensity factors should be included. It is only recommended that both location-based and market-based methods be used in reporting.

The emission factor of 0.60 kg CO_2e per kWh used in the example represents a worst case scenario. In 2022, the average emissions intensity in the EU was 251 g CO_2e per kWh, with large differences between countries [Eur23]. The best performer was Sweden with 7 g CO_2e per kWh, the worst was Poland with 666 g CO_2e per kWh. Germany was in the bottom third with 366 g CO_2e per kWh. The emission factors also differ greatly depending on the time of day and season, as these are influenced by the availability of sun and wind. The variability of emissions intensity by region and time makes precise GHG balancing practically impossible. If primary data can't be used or are not available, the ICT Sector Guidance recommends to use proxy data or estimated data and modeling methods (will be explained in the Section 2.6.3 "Proxy Metrics & Model-Based Estimations").

When assessing products in terms of the GHG Protocol, the entire life cycle must be considered. In addition to the actual use phase (scope 1 and 2), this also includes production, distribution, transportation and ultimately end-of-life treatment (scope 3). All emissions that occur outside the use phase are summarized as *embodied emissions*. The proportion of embodied emissions varies greatly depending on the product or service in the ICT sector. According to the ICT Sector Guidance, embodied emissions are highest for consumer devices such as personal computers or smartphones due to their comparatively short operating time. For devices with a long lifespan (e.g. network products) or devices that are heavily utilized (e.g. servers), emissions from use are higher than embodied emissions.

ICT products are often not used by one user alone. For example, a data center is used by a large number of users. A further challenge in GHG accounting is therefore to fairly calculate the total emissions (scope 1, 2 and 3) down to individual uses or users. This is referred to as *allocation*. Depending on the context, different allocation methods can be used for this. The next section lists possible methods that can be used specifically for software products.

Assessing GHG Emissions Related to Software

In the case of software companies and products, most emissions are generated in scope 2 and 3. Scope 2 includes the purchased electricity required for operation, scope 3 includes the indirect emissions caused by the manufacture of the devices used. According to the ICT Sector Guidance, the share of embodied emissions in the total emissions of software products is small, especially if the software is used a lot. This applies in particular to cloud applications, which are generally heavily used and share the hardware resources used with many others due to virtualization. The ICT Sector Guidance of the GHG protocol emphasizes that the energy consumption of ICT hardware is primarily influenced by software. The guidance states that software designers must have a comprehensive understanding of the energy consumption of their software in order to achieve efficient energy use.

The ICT Sector Guidance presents two methods for calculating GHG emissions attributable to software. The method presented as "Part A" comprises a full life cycle assessment covering all five phases: material acquisition and pre-processing; production; distribution and storage; use; and end of life. A full life cycle analysis is not relevant to this thesis, so this method will not be explained here. As "Part B", a more detailed method specifically for software developers is presented, with the aim of measuring the energy demand during the use phase of software in order to subsequently calculate the corresponding GHG emissions. The most important aspects of this method are presented in the Section 2.6 "Assessing the Energy Efficiency of Software".

2.5.5 GHG Reporting in the Cloud

The GHG accounting of software applications and services used in the cloud is particularly challenging. Without the provision of data by the cloud providers, accurate accounting is hardly possible.

All hyperscalers have started offering dashboards for customers to make the carbon footprint of their own cloud usage transparent: *AWS Customer Carbon Footprint Tool*,[37] *Emissions Impact Dashboard for Azure*,[38] *Google Carbon Footprint*.[39] The data is provided for the previous months and is therefore not available in real time. Accordingly, the dashboards are currently primarily used for reporting purposes and are unsuitable for carrying out measurements and potential optimizations at development or maintenance time. The methodology and transparency of the data used differs greatly between the cloud providers. According to the methodologies made available online at the time of this writing, only Google uses real energy consumption data to calculate the carbon

[37] https://aws.amazon.com/aws-cost-management/aws-customer-carbon-footprint-tool
[38] https://www.microsoft.com/en-us/sustainability/emissions-impact-dashboard
[39] https://cloud.google.com/carbon-footprint

footprint, while Azure estimates energy consumption based on utilization and AWS does not provide any information on the methodology. Google allocates machine energy usage to internal services by separately evaluating dynamic power (energy used during workload) and idle power [Goo24]. Hourly dynamic power is allocated to internal services based on relative CPU usage, while idle power is allocated based on resource allocation. Overhead energy consumption such as power systems, cooling and lighting is also included. Energy consumption for shared infrastructure services is reallocated based on relative usage or internal cost of service where data is insufficient. In terms of the carbon intensity values used for the energy consumed, only Google uses hourly updated, location-based carbon intensity values, while Azure and AWS use a market-based approach and therefore do not reflect the actual emissions generated. Scope 3 emissions are largely included by Google and Azure, while AWS has so far only covered Scope 1 and 2.

2.5.6 Cloud Carbon Footprint

Cloud Carbon Footprint (CCF)[40] is a popular open source tool that can estimate the GHG footprint of workloads in the AWS, Azure and Google clouds. It uses usage data (compute, storage, networking, etc.) to calculate estimated energy consumption and then converts this into GHG emissions using carbon intensity factors [Tho22]. In addition to operational emissions, estimated embodied emissions are also included. The available information from the cloud providers' billing data is used as the basis for the estimates. Typical energy coefficients are used for the energy estimates for the respective cloud resources used.

The following formulas are used by the CCF tool to estimate the energy consumption of a compute resource:

Average Watts = Min Watts + Avg vCPU Utilization × (Max Watts − Min Watts)

Compute Watt-Hours = Average Watts × vCPU Hours

Explanation of the parameters:

- *Min Watts* (constant): The SPECPower database is used to determine this constant depending on the assumed CPU.
- *Max Watts* (constant): Same as Min Watts.
- *Avg vCPU Utilization* (variable or constant): If provided by the cloud provider APIs or if unknown, an average utilization of 50 % is assumed (same value as described in the Section 2.5.2 "Efficiency of IT Components").
- *vCPU Hours* (variable): How long the resource has been in use.

For storage resources, CCF is using a similar formula based on coefficient values:

$$\text{Watts per Terabyte} = \frac{\text{Watts per disk}}{\text{Terabytes per disk}}$$

Depending on the storage type, Hard Disk Drive (HDD) or Solid Disk Drive (SSD), the estimated Watts per Terabyte are different:

[40] https://www.cloudcarbonfootprint.org

Watts per Terabyte (HDD) = $\frac{6.5\ W}{10\ TB} = 0.65\ \frac{Wh}{TBh}$

Watts per Terabyte (SSD) = $\frac{5\ W}{6\ TB} = 1.2\ \frac{Wh}{TBh}$

Replication factors are also applied to take account of the redundancy of data in cloud storage.

Data transfer over the network is only considered by the CCF among different geographical data centers. The CCF documentation states, that "it is safe to assume that the electricity used to power the internal network is close to 0, or at least negligible compared to the electricity required to power servers" [Tho22]. In addition, traffic leaving a data center to deliver services to end users is also excluded, as it is considered to be managed by Content Delivery Network (CDN) providers, which should be reported separately. Additionally, this outbound traffic is influenced by end-user behavior.

For data transfer between different geographically separated data centers, a constant is again assumed with which an estimate of the energy consumption can be made. Currently, at the time of writing this paper, the value 0.001 $\frac{kWh}{GB}$ is used. However, the methodology and the value are the focus of some discussion.[41,42,43,44]

For memory, the coefficient factor of 0.000392 $\frac{kWh}{GBh}$ is used.

CCF uses region-specific average values for the carbon intensity factors by default, i.e. it uses a location-based approach. It is also possible to query the carbon intensity in real time using the API of Electricity Maps[45] to obtain more accurate and up-to-date emission estimates.

The methodology used by the Cloud Carbon Footprint tool shows that only a rough estimate of the greenhouse gas footprint of cloud workloads is currently possible, as there is a lack of primary data and many assumptions have to be made.

2.5.7 Software Carbon Intensity Specification

The *Software Carbon Intensity (SCI)* specification defines a methodology for calculating emissions from software for a specific functional unit [Fou22]. The specification includes both the energy consumption and the embodied emissions of the reserved hardware resources. At the time of writing this thesis, the SCI specification is in the International Organization for Standardization (ISO)[46] standardization process. The specification was developed by the GSF,[47] a non-profit organization that has been growing steadily since 2021, with the mission to "build a trusted ecosystem of people, standards, tooling and best practices for creating and building green software" [Fou24].

The basic formula for calculating the SCI score is as follows:

$SCI = (E \times I) + M$ per R

[41] https://github.com/cloud-carbon-footprint/cloud-carbon-footprint/issues/379
[42] https://github.com/cloud-carbon-footprint/cloud-carbon-footprint/issues/723
[43] https://github.com/cloud-carbon-footprint/cloud-carbon-footprint/issues/951
[44] https://github.com/cloud-carbon-footprint/cloud-carbon-footprint/discussions/1045
[45] https://www.electricitymaps.com
[46] https://www.iso.org/standard/86612.html
[47] https://greensoftware.foundation

E is the energy consumption of the software components for a certain period of time (kWh), I the emission intensity factor ($\frac{g\ CO_2e}{kWh}$), M the embodied emissions ($g\ CO_2e$) and R the functional unit [Fou22]. Before the SCI score can be calculated, the system boundaries and the functional unit must first be defined. For example, the system boundaries can include the compute, storage and network resources used in the production environment, but exclude the resources on the user side. The functional unit should be defined in such a way that it describes well how the application scales. For example, if the application scales with the number of users, *user* should be selected as the functional unit. Other suggested functional units are *API call/request*, *benchmark*, *transaction* and *database read/write*.

The SCI aims to encourage the reduction of carbon emissions from software applications, to have an influence on a system-wide level and to make the implementation as simple as possible. The SCI score is well suited to quantifying the effects of software changes, for example measures to reduce emissions. The score should first be calculated for the software without changes and then with. The comparison then shows the extent to which the changes affect the carbon footprint of the software.

2.5.8 Standard Usage Scenario

Similar to the concept of a functional unit, there is the concept of a *standard usage scenario*. According to Kern et al. [KHG+18], the energy consumption of software should be measured on the basis of the typical use of the application under realistic conditions. For this purpose, the concept of a standard usage scenario was designed. To create one, Kern et al. recommend to identify the tasks that users typically perform within the application and the functionalities that require high energy or resource usage. From this information, a flow can be created that can then be automatically executed on a reference system. This procedure makes it possible to achieve reproducible measurements to the greatest possible extent.

The concepts of functional unit and standard usage scenario are both intended to relate the evaluation of a system to a relevant unit in order to be able to make useful conclusions. The functional unit, as defined by the GSF, focuses on scalability, while the concept of the standard usage scenario focuses on the typical use of the system. It is possible that both are the same. For example, for an online service, the functional unit *API call* may be appropriate, which may well correspond to a standard usage scenario for an API-focused service. However, for more complex applications, a standard usage scenario can encompass much more than a functional unit and, for example, cover a complete workflow in a word processing application.

2.6 Assessing the Energy Efficiency of Software

The previous section showed the importance of software energy consumption in the generation of GHG emissions in the ICT sector. This section now looks at the energy efficiency of software, how it can be measured and evaluated and how energy consumption can ultimately be reduced.

2.6.1 Definition of Energy Efficiency

The energy efficiency of a software describes the relationship between the functionality provided (useful work done) and the energy consumption [JDNK12]. Therefore, an energy-efficient software uses less energy to deliver the same value than less efficient software. Energy efficiency is a non-functional quality requirement for software, with the aim of reducing negative first-order impacts [PRRT14].

Energy efficiency is not yet considered as a common quality criterion in software architecture work. In the literature, however, more and more voices can be found calling for energy efficiency to be considered a first-class quality attribute [BCK21]. As is usual with quality attributes, trade-offs must be considered, as not all attributes can be fulfilled in their entirety. Energy efficiency, for example, competes with availability, buildability and modifiability [BCK21].

There is a certain link between the quality aspects of energy efficiency and performance. Often the two correlate, but there are exceptions that should be considered. For example, the use of additional server infrastructure can lead to improved performance because the workload can be distributed and more requests can be processed at the same time, while at the same time the total energy consumption increases due to the additional servers working in parallel. In addition, server idling has no negative impact on performance, but it does have a negative impact on power consumption. The section on Related Work takes a close look at the differences in performance between different architectures.

2.6.2 Measuring the Energy Consumption of Software

This section describes various methods and approaches for measuring the energy consumption of software, which is not always easy in practice.

Measurement Methods & Tools

Energy consumption can be captured using different methods. Power meters can be used to measure the exact energy consumption of all hardware components of a physical machine. However, this method requires additional hardware and is not applicable in all environments, e.g. virtualized cloud environments. A simpler variant that does not require additional hardware is to use existing hardware functionalities to provide data on energy consumption. *Running Average Power Limit (RAPL)* is one such functionality introduced by Intel in 2011 with the *Sandy Bridge* processor architecture. RAPL can provide data on the power consumed by the processor and main memory based on a model-based power estimation, which is comparatively easy to retrieve and quite accurate [KHN+18]. However, RAPL has the disadvantages that it can only be used with certain processors, it cannot be used to retrieve data in virtualized cloud environments, and it cannot be used to track the power consumption of other hardware components, such as a dedicated graphics card.

There are several tools that simplify the measuring of the energy consumption of a software application. Here is a list of some tools that use RAPL for measuring: *Intel Power Gadget*,[48] *PowerTOP*,[49] *PowerAPI*,[50] *JoularJX*,[51] *Scaphandre*,[52] *Green Metrics Tool*[53] and Kepler.[54]

If the use of a power meter or hardware functionalities such as RAPL is not an option, there is also the possibility of using estimations (see Section 2.6.3 "Proxy Metrics & Model-Based Estimations").

Application Isolation

Three different types of software must be taken into account when recording the power consumption of software: Operating System (OS), application and virtualization. These must each be prepared for a measurement and the conditions must be documented. The hardware used for the measurement must also be defined and documented in advance. If an exact measurement is not possible, which is often the case within virtualized environments, an estimate can be made using secondary data.

In measurements, the overall energy consumption is a composite of the energy used by both the operating system and the application(s). To differentiate between the energy usage of the operating system and the application(s), the *power subtraction* method can be applied [TI17]:

$$P_{application} = \overline{P_{Measure}} - P_{Idle}$$

$P_{application}$ is the power consumption of the application, $\overline{P_{Measure}}$ is the average power consumption observed during power measurement tests, and P_{Idle} is the average power consumption of the operating system when idle. Before the actual measurement of the power consumption of an application is carried out, the power consumption in idle must first be measured.

Process Allocation

Allocation methods are required to divide the total power consumption of a system between several applications. Typically, the CPU utilization is used for this, as the power consumption correlates with the utilization rate [TI17]. The ICT sector guidance of the GHG protocol explains the process as follows: To analyze the energy consumption of one application, here called *application 1*, start by computing the total processor utilization across all applications, e.g. 60 % from application 1 + 8 % from application 2, resulting in 68 %. Subsequently, determine the specific application's overall percentage of the measured processor utilization, like application 1's 60/68, yielding 88 %. With this utilization value (U) for application 1, set at 88 %, you can now calculate the proportional energy consumption of application 1:

$$P_{application} = U \times P_{all\_applications}$$

[48] https://www.intel.com/content/www/us/en/developer/articles/tool/power-gadget.html
[49] https://github.com/fenrus75/powertop
[50] https://powerapi.org
[51] https://github.com/joular/joularjx
[52] https://github.com/hubblo-org/scaphandre
[53] https://github.com/green-coding-berlin/green-metrics-tool
[54] https://github.com/sustainable-computing-io/kepler

In this context, $P_{application}$ signifies the energy consumption of application 1, U stands for the utilization value (e.g. 88 %), and $P_{all\_applications}$ denotes the total energy consumption across all applications.

The various tools each have their own philosophies on whether and how the process allocation is made:

- Kepler uses process-level CPU utilization based on CPU instructions (is configurable).
- Scaphandre uses process-level CPU utilization based on CPU time.
- PowerAPI uses performance based regression modeling.
- Green Metrics Tool doesn't support per-process measurement, because it follows the philosophy of usage scenarios that should contain all components to reflect actual use cases of the software.

Virtualized Environments

Virtualized software runtime environments, such as virtual machines or containers, pose a further challenge to energy efficiency assessment. The capturing of the actual energy consumption of software applications located within a virtualized environment is only possible with additional software on the host system. Software on the host system can track the usage of all hardware resources – e.g. CPU cores, disk arrays, memory banks, network cards and graphics cards – to determine at what power level the resource was operating and for which virtual machine it was working [KZL+10]. On a developer workstation or in-house servers this approach can be used for measurement. However, this is not the case for servers in cloud computing environments. In virtualized cloud environments, it is up to the cloud operator to provide the necessary data. However, none of the major cloud operators provide data on energy consumption.

The ICT Sector Guidance of the GHG Protocol describes estimation methods using use profiles for virtual environments [TI17]:

- *Static-use profile* serves as an educated estimate or measured average value indicating the probable utilization of a device's resources in relation to its capabilities. This method, offering a straightforward and rapid approach, may be informed by insights from data center operators and hardware vendors based on server hardware components' utilization patterns.
- *Dynamic-use profile* involves monitoring a device's activity directly or relying on measurements derived from likely Virtual Machine (VM) behavior, such as application load test cases or network traffic traces. This profile proves crucial for empirical measurements of device power consumption, allowing a real-time assessment of the device's behavior and providing valuable data for accurate power consumption measurements.

Organizations like the Transaction Processing Performance Council (TPC),[55] the Standard Performance Evaluation Corporation (SPEC),[56] and the Storage Performance Council (SPC)[57] offer widely-used use profiles for web applications. These established standards assist in benchmarking and evaluating the performance of web applications across various metrics.

Within a VM, it is not possible to see how many other VMs and applications are currently running on the same physical server. It is therefore not clear how heavily a CPU is actually utilized and how a use profile, especially a non-linear, dynamic use profile, can be applied for the estimation. Without further information from the operator, such estimates therefore always remain a rough approximation.

2.6.3 Proxy Metrics & Model-Based Estimations

Depending on the environment, measuring the energy consumption of a software can be difficult, costly or even impossible (e.g. in virtual environments). Proxy metrics and models can help to estimate the energy efficiency of software regardless. Instead of measuring the actual energy consumption, other metrics that correlates with energy consumption can be used for estimations.

Basic proxy metrics that can be used to assess energy efficiency are CPU utilization, memory utilization, network throughput and disk throughput [JVB+17]. These metrics can be used to roughly estimate the efficiency of a system for the respective factor, but not to estimate a value for the total energy consumption.

As seen in the example of the Cloud Carbon Footprint tool, energy co-efficiency values can be used to estimate energy consumption with the help of utilization data and a simple formula. However, these are only a rough approximation and assume linear energy consumption. Machine learning models, on the other hand, are more accurate.

Machine learning models often use the data from the *SPECpower_ssj2008 benchmark*[58] to accurately estimate energy and performance values of a particular server configuration [RBKW22]. Projects in practice that use this approach are *Cloud Energy*[59] by Green Coding Solutions and *Kepler*.[60] Measurements show that these work significantly better than other estimation methods. Kepler's power model has a significantly reduced mean squared error (MSE) of accuracy, measured only at 0.010 [ACC+23]. This is in contrast to alternative approaches, where the MSE rises to 0.16 when using a simplified ratio method, and further increases to 0.92 when using aggregated workloads for model training.

[55] https://www.tpc.org
[56] https://www.spec.org
[57] https://www.storageperformance.org
[58] https://www.spec.org/power_ssj2008
[59] https://www.green-coding.io/projects/cloud-energy/
[60] https://sustainable-computing.io/design/power_model

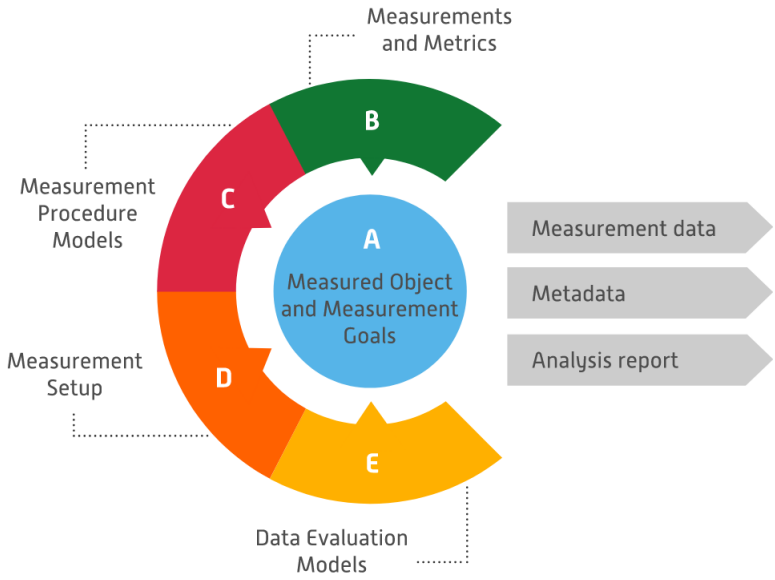

Figure 2.4: Components of the Green Software Measurement Model. Source: [GBC+24]

2.6.4 Green Software Measurement Model (GSMM)

In early 2024, Guldner et al. [GBC+24] published a comprehensive paper summarizing the core ideas of the measurement models, setups and methods in relation to energy and resource consumption of over 10 research groups in the form of a model. They have developed a model from this, which they call the *Green Software Measurement Model (GSMM)*.

Figure 2.4 illustrates the components of the GSMM. The individual components are [GBC+24]:

- *(A) Measured object and measurement goals*, e.g. comparison of a software entity with itself within a Continouos Integration (CI) pipeline, between releases, or when introducing new features, or comparisons between different implementations, libraries, configurations, etc.
- *(B) Measurements and metrics*, in terms of energy efficiency, a metric must be chosen that is considered to be useful work (similar to the functional unit of the SCI score).
- *(C) Measurement procedure models* define the measurement method, including aspects like repetitions, which tools to use and which scenarios to devise. Scenarios can be classified in different ways, either by technique into white-box and black-box measurements, or by kind into idle, standard usage, load scenarios and baselines.
- *(D) Measurement setup* defines the necessary hardware and software stack for the measurement using the specifications from the measured object, metrics, and measurement method.
- *(E) Data evaluation models* defines the methods to analyze and evaluate the measurement data and prepare the measurement report.

Guldner et al. list various existing measurement methods and apply the GSMM to each of them. One of these is the Green Metrics Tool (GMT), created by Green Coding Solutions. GMT can be used to measure any type of software (A), as long as the software can be containerized. GMT can measure

a variety of metrics (B) using so-called “metric-providers”, e.g. CPU energy consumption using RAPL. GMT uses usage scenarios as the measurement procedure model (C). Only a Unix-based system is required for the measurement setup (D). Green Coding Solutions offers a measurement cluster, which is fully set up and can be used free of charge with restrictions. For analyzing the data (E), GMT provides a dashboard that graphically displays the metrics.

3 Related Work

Comparable studies that specifically looked at the differences in energy efficiency between monolith and microservices architectures could not be found. The search was carried out using Google Scholar and the following search pattern:

```
monolith AND microservices AND ("energy efficiency" OR "energy consumption" OR "power consumption")
```

As performance often correlates with energy efficiency, the search was extended accordingly:

```
monolith AND microservices AND (performance OR latency OR throughput)
```

This search has produced some interesting studies, which are discussed below. Many of the studies were conducted in the context of migrating from monolithic systems to microservices systems. For this reason, it was obvious to expand the search and look specifically for the implications of microservices architectures on performance:

```
microservices AND (performance OR latency OR throughput) AND ~implications
```

3.1 Monolith vs. Microservices in Terms of Performance

The following section discusses studies that looked specifically at the differences in performance between a monolithic and a microservices architecture. The studies are sorted according to their publication date.

Ueda et al. [UNO16] compared the performance of microservices with the monolithic architecture using the Acme Air benchmark suite,[1] which is available in both architectural styles and also with two different runtimes, Node.js and Java EE. The study came to the conclusion that the monolith performs significantly better and that the difference in performance increases with the granularity of the microservices. On the same hardware configuration, the performance of the microservice variant was 79 % lower than that of the monolithic variant. The authors observed that the microservices in the runtime libraries consumed significantly more time to process a client request than the monolith, namely 4.22x on a Node.js application server and 2.69x on a Java EE application server. The study only considered vertical scalability of the systems (1, 4 and 16 CPU cores), but not horizontal scalability. The Acme Air benchmark suite used has not been further developed for nine years, so it is not suitable for a new analysis, e.g. with regard to energy efficiency.

[1] https://github.com/acmeair

Villamizar et al. [VGO+17] compared the costs of using the cloud to run web applications with the different architectural styles monolithic, microservices and serverless (AWS Lambda[2]) and also examined the response times. The monolithic architecture was implemented with two tech stacks, one with the Java-based Play web framework[3] and one with Jax-RS,[4] the microservices application with Java and Play and the Lambda applications with Node.js.[5] The reference applications used are not available as open source. As part of this study, various test scenarios were carried out in the AWS Cloud and scaling effects were taken into account. The maximum number of requests per minute for the monolithic variant was used as the starting point for the tests. The study concludes that infrastructure costs can be significantly reduced with microservices and in particular with cloud environments specially designed for fine-grained architectures such as AWS Lambda. In the tests, the infrastructure costs for microservices were 13 % lower compared to monolithic applications, and with AWS Lambda the infrastructure costs could even be reduced by 77 %. However, the tests also showed that the response time increases significantly with the microservices architecture compared to the monolithic architecture, as each request has to pass through several components in the system. With Lambda, the response times were comparable to the monolithic architecture.

Al-Debagy and Martinek [AM18] came to the conclusion in their tests with generated Spring Boot applications[6] using JHipster[7] that the two architectures have similar performance values under average load. Under low load and concurrency tests, the monolith performed slightly better than the microservices. However, the tests were carried out exclusively in a local environment, whereby neither vertical nor horizontal scaling effects were evaluated.

Bjørndal et al. [BAB+21] have benchmarked a minimalistic library system that includes four use cases and considers synchronous and asynchronous relationships between microservices. The reference application used is implemented with ASP.NET Core[8] and is available as open source.[9,10] The tests were carried out with Docker and Kubernetes both in a local environment and in the Azure Cloud. An auto-scaler for the Kubernetes pods was used in the Azure Cloud. As a final result, the authors found that the monolith performed better than the microservices system in terms of throughput and latency, but not in terms of the scalability metric. Scalability was calculated based on the ratio between the increase in concurrent users and the resulting increase in hardware performance. This ratio was more than six times higher for the microservice system in the Azure Cloud than for the monolithic system. From this, the authors conclude that the microservice system scales significantly better with increased hardware resources. At the same time, however, the response times of the microservice system were always higher and the throughput lower than that of the monolithic system.

Blinowski et al. [BOP22] compared the performance and scalability of a monolithic and a microservices architecture using a self-developed web application both in a local and in the Azure Cloud. Both architectures were implemented with Java and Spring Boot and with C# and

[2] https://aws.amazon.com/lambda
[3] https://www.playframework.com
[4] https://projects.eclipse.org/projects/ee4j.rest
[5] https://nodejs.org
[6] https://github.com/eugenp/tutorials/tree/master/jhipster-modules
[7] https://www.jhipster.tech
[8] https://asp.net
[9] https://github.com/NichlasBjorndal/LibraryService-Monolith
[10] https://github.com/NichlasBjorndal/LibraryService-Microservice-DotNet

ASP.NET Core, so that ultimately four variants were examined. The applications are published open source in a GitHub repository.[11] The study revealed several interesting results. Firstly, it showed that the monolithic systems on a single machine perform better than their microservice-based counterparts. Due to the overhead of network communication, microservices in this scenario have poorer throughput and increased response time compared to the monolithic architecture. However, with increasing user requests, microservices can compensate for the overhead through better scaling and can perform significantly better than the monolith with larger user requests. However, according to the authors, it is not sufficient to compare performance alone when making comparisons in the cloud with different configurations. Instead, only configurations with similar infrastructure costs should be compared. In this respect, the monolithic reference applications performed slightly better than the microservices systems. In terms of scalability, the study also revealed interesting results. The experiments showed that in the Azure Cloud, vertical scaling is more cost-efficient than horizontal scaling in order to achieve the same performance. However, vertical scaling is limited by the most powerful cloud instance available. If this is not sufficient to process the number of requests, horizontal scaling has to be used. In the experiments, horizontal scaling scored particularly well in the compute-intensive application scenario, where the throughput was able to scale almost linearly with the number of instances. In the simple application scenario, however, the best performance was achieved with a small number of cloud instances. A larger number of instances actually caused the performance to degrade. According to the authors, this effect of overscaling occurs when the CPU overhead resulting from load balancing exceeds the benefits of increasing overall processing power. In their summary, Blinowski et al. conclude that a microservice architecture is not the best choice for every context and that a monolithic architecture seems to be the better choice for simple, small systems that do not need to support a large number of concurrent users.

Okrój and Jatkiewicz [OJ23] have investigated the differences in performance, scalability and costs when using microservices and monolithic architecture using a small self-developed Spring Boot application in the Azure Cloud. In almost every test they conducted, the monolithic architecture proved to be superior. However, their tests showed that in the Azure Cloud, microservices can be scaled horizontally in smaller steps than vertically, potentially saving money. However, the study is very small and is limited to horizontal scaling to a maximum of three *B1* machines in the Azure Cloud.

Faustino et al. [FGPR24] carried out a step-by-step migration from a relatively complex monolithic application called LdoD[12] with 71 domain entities to a microservices architecture and evaluated the performance and migration effort. As an intermediate step, they first created a modular monolith from the monolith, so that only the inter-module interfaces had to be adapted during the subsequent migration to a microservices architecture. When comparing the performance of the modular monolithic architecture and the microservice architecture on one machine without scaling, they found that the microservice architecture had significant negative effects in almost all tested scenarios, both in terms of latency and throughput. The authors attribute the performance degradation to the number of remote calls and the associated latency due to network overhead and the time required for data serialization and deserialization. The only exception in which the microservices architecture had no negative impact on performance in the test scenarios was a computationally intensive scenario. On the one hand, the compute-intensive scenario reduced the impact of distributed communication

[11] https://github.com/annaojdowska/monolith-vs-microservices
[12] https://github.com/socialsoftware/edition

on overall performance and, on the other hand, optimizations in the microservices architecture, such as the use of caches, had a positive impact on performance. However, it is also noted that caches can have a negative impact on data consistency, so their use and configuration must be considered on a case-by-case basis. Faustino et al. also tested the microservices application for scalability using the Google Kubernetes Engine.[13] This demonstrated the advantages of microservices, which showed a significant improvement in throughput through scaling. However, the latency values were still significantly high, especially with large amounts of information. To summarize, the authors conclude that in this case study, the migration to a microservices architecture had a severe impact on performance due to the network overhead, which proved to be far too high compared to previous architectures, but provided a more scalable and maintainable architecture.

Finally, a brief summary of the points on which the above study results largely coincide: - Monolithic architectures have advantages at low load, both in terms of throughput and cloud resource costs. - Microservices architectures can enable better throughput under high load and especially in compute-intensive scenarios, as they can exploit their advantages of better scalability here. - Microservices architectures tend to have higher response times due to the network communication required between individual components.

Based on the results discussed, which architecture is more suitable in terms of performance and costs depends on the individual requirements in terms of the expected load and the potentially required scalability and cannot be answered in general terms.

3.2 Further Microservices Performance Implications

Gan et al. [GZC+19] have conducted a comprehensive analysis of how the microservices architecture style affects various aspects of the cloud system stack. Several representative microservices reference systems from the *DeathStarBench suite*[14] were used as a basis. Comparisons were also made with monolithic applications. The study shows in particular the important role played by network communication in microservices systems. Compared to monolithic systems, microservices systems spend significantly more time processing network communication. Another interesting finding is that microservices take longer to recover from a QoS violation than monolithic applications. This is the case even when automatic scaling mechanisms are used. According to the authors, this is due to the fact that greater latencies in a system can quickly affect upstream services. Accordingly, the paper concludes that the more complex the graph of an application's microservices is, the greater the impact of slow services. With monoliths, on the other hand, only one instance is affected by a slow server, but not other instances. In order to minimize these risks with microservices, the authors recommend having proper cluster management in order to adequately meet the different requirements of the individual services.

[13] https://cloud.google.com/kubernetes-engine
[14] https://github.com/delimitrou/DeathStarBench

4 Reference Application

This chapter describes the architecture of the reference application, which decisions were made how and why, and what differences there are between the two variants T2-Modulith and T2-Microservices.

The structure of this chapter is inspired by the arc42 template, but omits some parts that are not in the scope of this thesis. The chapter begins with the initial decision-making process that led to the decision to use the T2-Project as the basis for this thesis. Next, the architecture of the two variants is documented.

4.1 Initial Decision-Making Process

As already mentioned in the introduction chapter in the Section 1.3, at the start of the thesis the decision was made to use the already existing microservices reference application T2-Project as the basis for providing the two required variants. The T2-Project was developed about two years earlier at the same institute where this work is being written. The initial purpose of the T2-Project was to test the microservices architecture style in cases of Service Level Objective (SLO) violations with regard to response time and availability.

On the way to the decision to use the T2-Project as a basis, several alternative options were considered. The development of a custom reference system in two different variants is unrealistic within the timeframe of a master's thesis. An existing and fully functional open source reference system, which is available in both a microservices and a monolith architecture and with a similar tech stack, would have been ideal, but could not be found during an initial search. The Google project "monolith-to-microservices-example"[1] was found that comes close to this, but it has some disadvantages: it uses three different programming languages for the microservices (Java, Python and Go) and Java for the monolith, that may influence the comparison in a negative way, it is very minimalistic, e.g. data integrity is not guaranteed, and it is outdated (microservices variant is further developed in the separate repository microservices-demo,[2] but not the monolith variant).

There are a large number of reference systems for the microservices architecture style alone, as the GitHub repository "davidetaibi/Microservices_Project_List"[3] shows.

[1] https://github.com/GoogleCloudPlatform/monolith-to-microservices-example
[2] https://github.com/GoogleCloudPlatform/microservices-demo
[3] https://github.com/davidetaibi/Microservices_Project_List

The following advantages of the microservices reference system T2-Project were identified in comparison to potential alternatives:

- With Spring Boot, the T2-Project has a comparatively modern tech stack that is already known.
- The use of the saga pattern is a unique characteristic among the reference systems, which makes the comparison with a monolithic system that does not require distributed transactions more interesting.
- The project originates from the same institute, making it easy to establish personal contact with the original developers.
- The project can be taken over in its entirety and developed further.

However, the T2-Project also has disadvantages:

- The microservices have not been developed in a year, and an outdated version of the Spring Framework (version 2.4.3) is being used, so dependencies need to be updated and adjustments made.
- The project has not yet been widely used, so that hardly any experience has been gained as to the extent to which the T2-Project reflects a realistic scenario and how suitable it is as a reference system. In addition, there may be some bugs that occur that need to be fixed. In fact, it is very likely that no one else is currently using the project.

In the end, the decision was made in favor of the T2-Project due to the advantages mentioned above. Upgrades and minor changes that are required to the T2-Project should be made relatively quickly. The disadvantage that the T2-Project has not yet been widely used can potentially have a negative impact on the validity of the results of the experiments, but otherwise does not affect the thesis.

At the end of the thesis period, an extended literature search on related work and performance comparisons (see Chapter 3 “Related Work”) did reveal reference systems that could have met the requirements for this thesis. These are not relevant for carrying out own tests in the context of this thesis, but should be mentioned here for the sake of completeness. For their study, [BAB+21] implemented a minimalist library software with four use cases in both a monolithic and a microservices variant based on the ASP.NET Core Framework. The source code was published as open source in two GitHub repositories, one for the monolithic variant[4] and one for the microservices variant.[5] The reference application was only used for this one study and has not been further developed in 5 years. [BOP22] implemented six variants for their study, two of them as a monolith, four of them with the microservices architecture. Both architectures were implemented in Java and Spring Boot as well as C# and ASP.NET Core. In addition, two further variants were implemented for the microservices system, which use gRPC[6] instead of the Hypertext Transfer Protocol (HTTP) for communication. The six variants are available as open source in the GitHub repository “annaojdowska/monolith-vs-microservices”.[7] The applications were only used for this

[4] https://github.com/NichlasBjorndal/LibraryService-Monolith
[5] https://github.com/NichlasBjorndal/LibraryService-Microservice-DotNet
[6] https://grpc.io
[7] https://github.com/annaojdowska/monolith-vs-microservices

one study and have not been further developed for 4 years. [FGPR24] have migrated an existing monolithic software called LdoD to a modular monolith and a microservices system for their study. The monolith is in productive use[8] and serves as a digital archive for the digital humanities. The tech stack consists of Java and Spring Boot. All three variants are publicly accessible in the GitHub repository "socialsoftware/edition",[9] each via its own branch: `master` contains the code for the monolith and `modular-monolith` and `microservices` the code for the architecture style of the same name.

4.2 Goals & Constraints

A reference system is required in two variants, one in a modular monolithic architecture (T2-Modulith) and one in a microservices architecture (T2-Microservices), which do not differ functionally from one another. The T2-Project is aimed at researchers and practitioners who want to test certain architectural decisions with regard to desired quality aspects. Originally, the focus was on the quality aspects of response time and availability for SLO violations. With this thesis, the T2-Project is extended for tests related to energy efficiency.

Requirements for the provision of the two variants are:

- Identical functionality and identical interface to the outside world.
- Compliance with a modular structure in the sense of DDD.
- Differences arise solely from the architectural differences between the monolith and microservices styles in order to ensure comparability. This means, for example, that the underlying tech stack must be identical.
- Data consistency must always be guaranteed.
- Existing user documentation[10] needs to be updated and documentation for the monolith variant needs to be added.

4.3 System Scope and Context

The T2-Project is a fictitious store for tea and provides a minimalist User Interface (UI) that enables an ordering process (without real payment). However, as a reference application, the T2-Project is primarily aimed at researchers who want to carry out various tests. The UI is not required for this, as the tests are carried out directly via the API provided. The interface of a credit institute is simulated for the payment process. The payment process can also be completely deactivated via configuration so that the fake credit institute is not necessarily required for tests. Therefore, there are no dependencies on real external systems. Figure 4.1 visualizes this as a system context diagram.

[8] https://ldod.uc.pt/?lang=en
[9] https://github.com/socialsoftware/edition
[10] https://t2-documentation.readthedocs.io

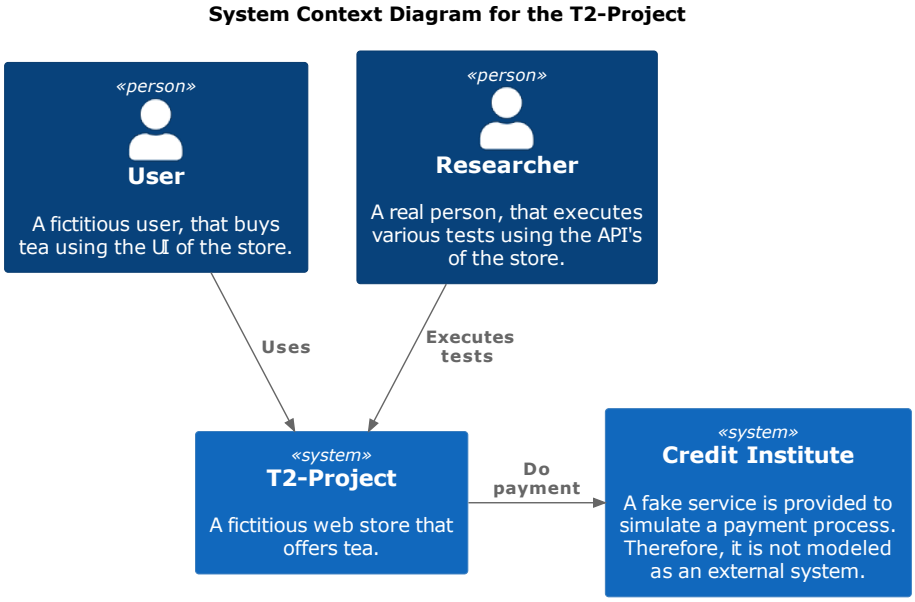

Figure 4.1: C4 system context diagram for the T2-Project

4.4 Solution Strategy

To provide the microservices variant (T2-Microservices), only slight adjustments had to be made to the original T2-Project. The source code of the microservices was used to create the second variant of the modular monolith (T2-Modulith). The following architectural approaches were used to achieve the goals and requirements:

Table 4.1: T2-Project main goals and architectural approach.

Goal / Requirement	Architectural Approach	Details
Same functionality	Source code of the existing microservices is copied to the monolith including existing unit tests. The UI component, based on Jakarta Server Pages (JSP), is also integrated into the monolith.	Section 4.9.2
Modularity	The structure of the existing microservices variant is already consistent with bounded contexts. Therefore, the module structure is copied to the monolith. To ensure that the module limits are always adhered to, the library *Spring Modulith* is used.	Section 4.8.1
Comparability	Same tech stack is used: Java, Spring Framework, MongoDB, PostgreSQL, etc. All dependencies are upgraded to the latest version or the latest Long-Term Support (LTS) version.	Section 4.5
Data consistency	Microservices variant already uses the saga pattern to ensure data consistency. The monolith variant can just use local transactions.	Section 4.8.4

Table 4.1: T2-Project main goals and architectural approach.

Goal / Requirement	Architectural Approach	Details
User documentation	The existing user documentation for the T2-Project had the focus only on the microservices variant. Therefore, the structure is split into two main parts, “Microservices” and “Modulith”, and a new part “Measurements” is added for documenting some user relevant details regarding energy measurements.	

4.5 Building Block View

This section describes the decomposition of both T2-Project variants. The C4 model is used for visualization. Level 1 shows the high-level building blocks, level 2 shows the internal structure of the T2-Modulith and it shows how the messaging is done for saga within the T2-Microservices system.

4.5.1 Level 1

The T2-Microservices system consists of a total of seven microservices, four databases and the optional *CreditInstitute* microservice. Figure 4.2 illustrates how the individual services and databases are connected to each other and which technologies are used.

The seven microservices have the following functions:

- UI: The application frontend.
- UIBackend: API Gateway for the UI.
- Cart: Manages the user shopping carts. It saves the cart contents to the *Cart Database*.
- Orchestrator: Manages the saga. Saga data is stored in the *Saga Database*.
- Order: Persists orders to the *Order Database* and marks them as either *success* or *failure*.
- Payment: Handles the store’s payment by contacting the credit institute.
- Inventory: Manages the store’s products. They are stored in the *Inventory Database*.

All microservices are based on Java and the Spring Framework with Spring MVC and Spring Boot. The database technologies used are PostgreSQL as a relational database and MongoDB as a document-oriented database. The UI is implemented using the server-side technology JSP. The UIBackend microservice communicates over HTTP with a REST like interface. However, the endpoints don’t support Hypermedia as the Engine of Application State (HATEOAS). The saga participants, highlighted with a purple background color, use message based communication among each other. This is necessary because of the saga pattern. Figure 4.3 illustrates how the individual components are involved in message-based communication.

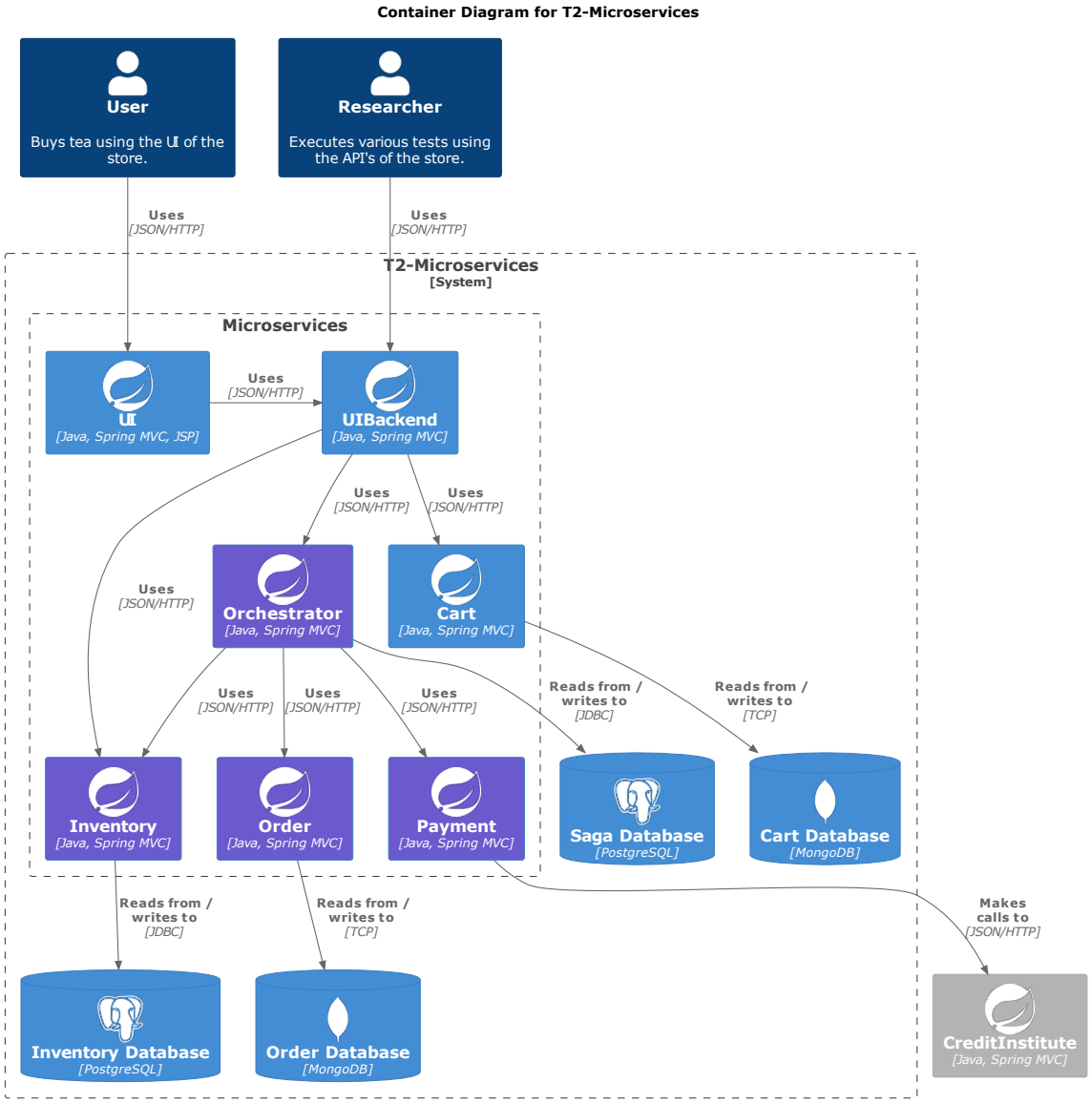

Figure 4.2: C4 container diagram for T2-Microservices

To implement saga, the frameworks *Eventuate Tram*[11] for transactional messaging and *Eventuate Tram Sagas*[12] for the orchestration of saga are used. Both work on top of Spring Boot. T2-Microservices uses the Eventuate Tram Core framework with Kafka as the message broker and PostgreSQL as the database. Eventuate provides slightly enhanced Docker images with custom configurations for both PostgreSQL[13] and Kafka.[14] The PostgreSQL image, for example, automatically creates the required database schemas on startup. Another component required for

[11] https://github.com/eventuate-tram/eventuate-tram-core
[12] https://github.com/eventuate-tram/eventuate-tram-sagas
[13] https://github.com/eventuate-foundation/eventuate-common/tree/master/postgres
[14] https://github.com/eventuate-foundation/eventuate-messaging-kafka

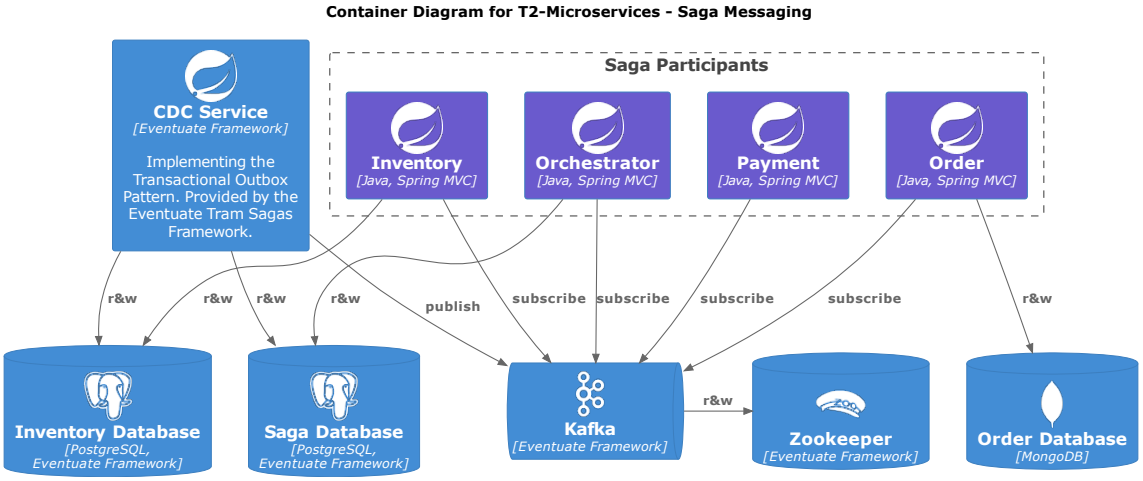

Figure 4.3: C4 container diagram for T2-Microservices – Saga Messaging

saga is the *Eventuate CDC Service*.[15]It implements the transactional outbox pattern, i.e. it handles the publishing of messages to Kafka when saga participants write to the saga database or the inventory microservices write to its database. The CDC reader uses PostgreSQL's write-ahead logging[16] for this purpose. It has been shown that this makes the saga run significantly faster than with polling. The abbreviation stands for Change Data Capture (CDC).

At this point, a problem in the implementation of the saga pattern becomes visible: The CDC service only implements the transactional outbox pattern for the PostgreSQL instances, but not for the MongoDB instance of the order microservice. As no transactions are configured for the MongoDB connection anyway, the transactional outbox pattern would have no use in this case anyway. This ultimately means that there is no transactional integrity in the operations of the order microservice with the current implementation.

How the saga works in detail is discussed in the Runtime View section.

In the T2-Modulith variant, the seven microservices are replaced with a single application, the *Modulith*, as shown in Figure 4.4. The modulith is based on the same tech stack and also integrates the UI, which is based on JSP. Only two databases are used, as no saga is required and a MongoDB database is used for both the shopping cart and the orders.

4.5.2 Level 2

It is now worth taking a closer look at the inner structure of the modulith. Figure 4.5 shows the inner module structure, which is very similar to the structure of microservices. The differences are that there is no longer an orchestrator, as no saga is required, and the order module takes on a more central role. The role of coordinator of the order confirmation process, which was previously held by the orchestrator, is now taken over by the order module. This decision was made due to the appropriate domain relationship. Section 4.8.1 discusses how the usage of *Spring Modulith* ensures

[15] https://eventuate.io/docs/manual/eventuate-tram/latest/cdc-configuration.html
[16] https://www.postgresql.org/docs/current/wal-intro.html

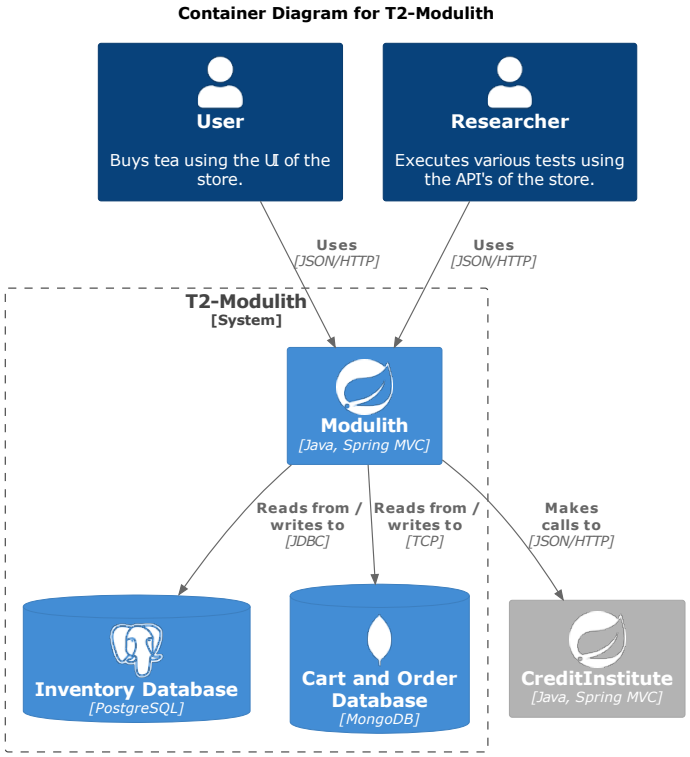

Figure 4.4: C4 container diagram for T2-Modulith

that the module boundaries (correspond to bounded contexts) are always maintained despite the monolithic architectural style. Spring Modulith can automatically generate a component diagram that corresponds to the one in Figure 4.5 except for the representation of the databases.

A look into the individual services of the microservices variant would not provide any added value at this point, so this is not included.

4.6 Runtime View

In this section, the typical workflow of an order will be examined with the help of sequence diagrams and, in particular, the differences between the two variants of the T2-Project will be worked out. The aspect that communication between the microservices takes place over the network via HTTP and between the modules in the monolith via inter-process communication is not discussed in detail here, but should be taken into account.

A typical T2-Project user scenario consists of three steps:

1. Get a list of all products
2. Add one or multiple items to cart
3. Proceed to checkout and confirm order

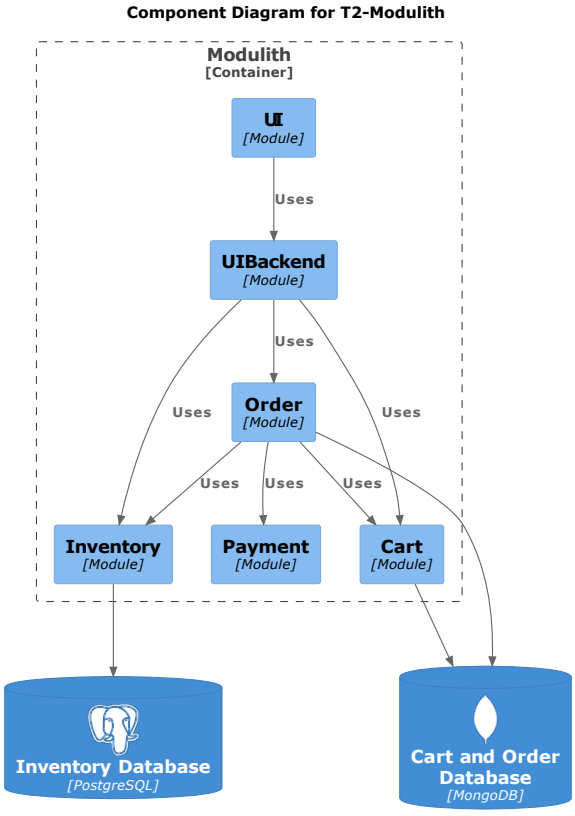

Figure 4.5: C4 component diagram for T2-Modulith

Figure 4.6 shows the sequence diagram for the processing of these three steps in the modulith in the case that only one item is added to the cart. To simplify the diagram, only the success case is shown, so that if conditions are omitted from the diagram. The first step, *Get Products*, simply retrieves all available products from the Inventory module. The second step, *Add Item to Cart*, first creates a reservation for the requested item before adding it to the cart. Only if there are enough items available, the item will be added to the cart. The order completion in step 3 is of more interest, as it differs from the T2-Microservices implementation:

1. *calculate total*: Order module calculates the total amount of all items in the cart
2. Get cart from the cart module
3. Get product details for each item in the cart
4. Calculate the total
5. *create order*: Order module creates a new order in the database
6. *do payment*: Payment module handles the payment and requests the external CreditInstitute service for this purpose
7. *commit reservations*: Inventory module commits the reservations previously created when items were added to the cart
8. `delete cart`: Cart module deletes the cart

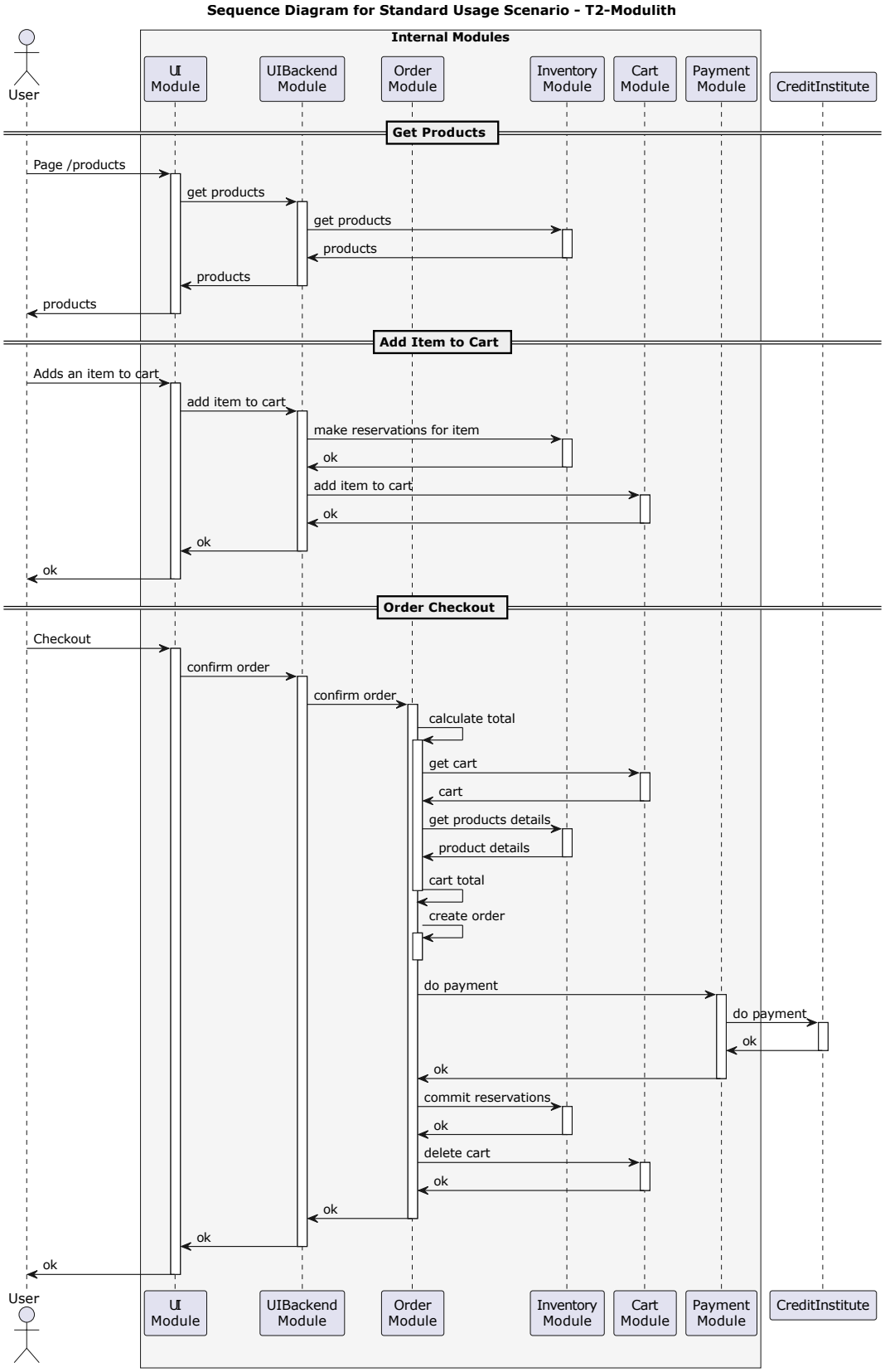

Figure 4.6: Sequence diagram for the standard usage scenario using the T2-Modulith variant

In the microservices variant, the sequence of the first two steps, *Get Products* and *Add Items to Cart*, is identical. However, the order checkout is divided into two parts in the microservices system: a synchronous part that starts the saga operation and an asynchronous part consisting of the saga execution, coordinated by the Orchestrator service. The synchronous part of the process is shown in the sequence diagram in Figure 4.7, the asynchronous part in Figure 4.8.

Another difference between the two variants can be seen in the synchronous part in the calculation of the total amount of the shopping cart (*cart total*): The microservices variant retrieves the product details from the Inventory service separately for each individual element in the shopping cart, whereas with the modulith this is done via a single call to the Inventory module. REST is used for the microservices, so the API of the Inventory service has been defined in such a way that either one product can be queried via its ID or all products. A search function via the API is not implemented. When implementing the monolithic system, the decision was made not to adopt this flow 1:1, but to implement it in a more practice-oriented way. As a result, the Inventory module provides a dedicated method for retrieving a list of products so that only one database query is required.

The sequence diagram in Figure 4.8 illustrates the asynchronous saga process in the success case, i.e. all local transactions are successful and no compensations are necessary. Kafka is used as a message broker for the communication. The CDC service, which is responsible for implementing the transactional outbox pattern, is omitted from this sequence diagram for simplicity. The role of the CDC service has already been covered in the section Level 1 as part of the Building Block View.

The order of the individual asynchronous saga operations corresponds to the chronological order of synchronous processing in the monolithic system. Table 4.2 shows how the saga is defined in the orchestrator and which compensation measures are defined. The *do payment* step is the pivot transaction of the saga, which means that if this step is successful, the saga will be completed. The *commit reservations* step is designed to be repeatable and succeed eventually. In the event of an error, the previously performed local transactions are compensated. The creation of an order object in the database is compensated for by setting the status of the order to *failure*. The shopping cart is already deleted after the order has been successfully submitted to the orchestrator as part of the synchronous procedure. This means that in the event of an error during the saga, the reservations must also be canceled (see step 1 in Table 4.2).

Table 4.2: T2-Project saga definition.

Step	Service	Transaction	Compensation
1	Inventory		cancel reservations
2	Order	create order	reject order
3	Payment	do payment	
4	Inventory	commit reservations	

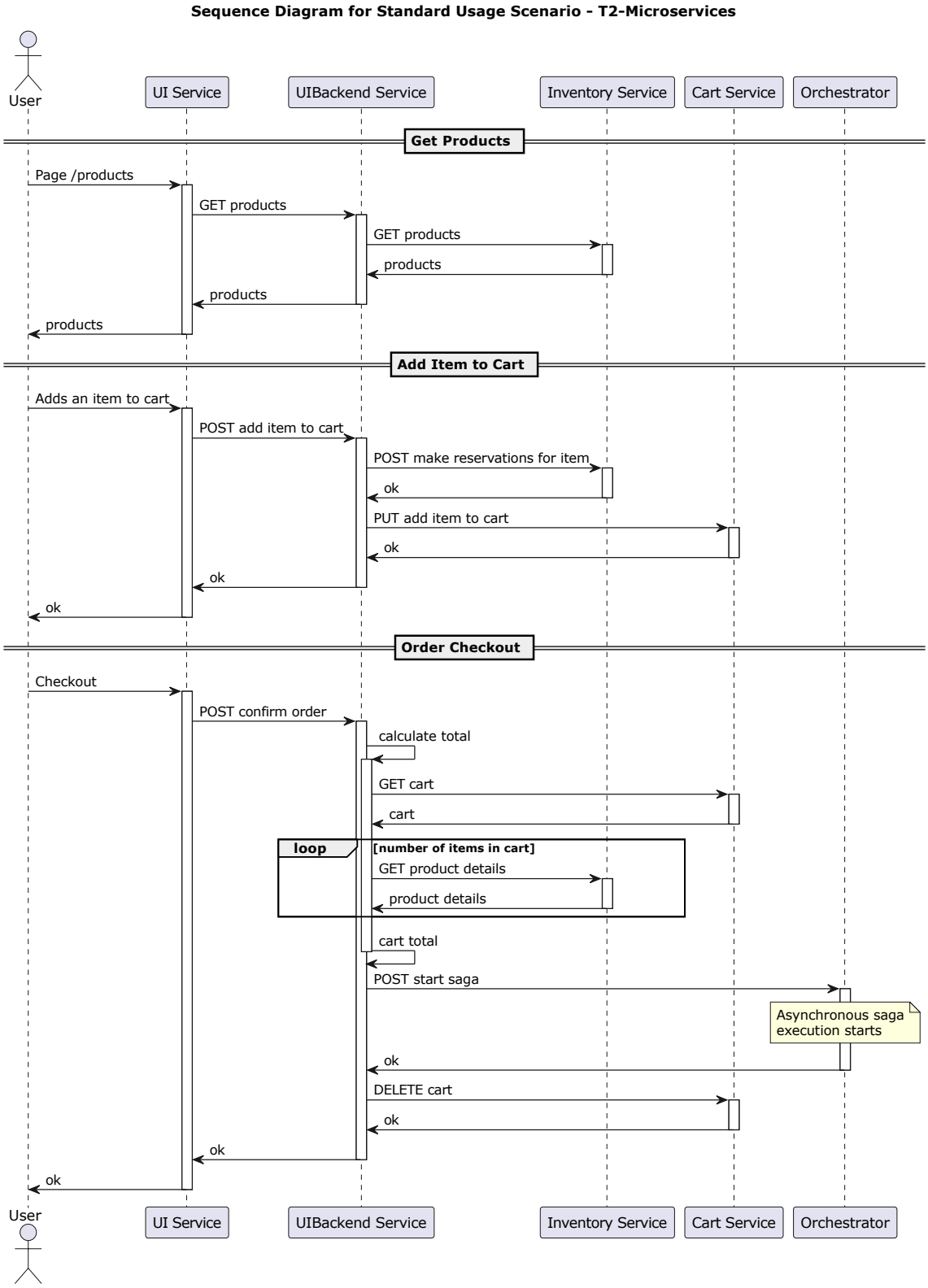

Figure 4.7: Sequence diagram for the standard usage scenario using the T2-Microservices variant

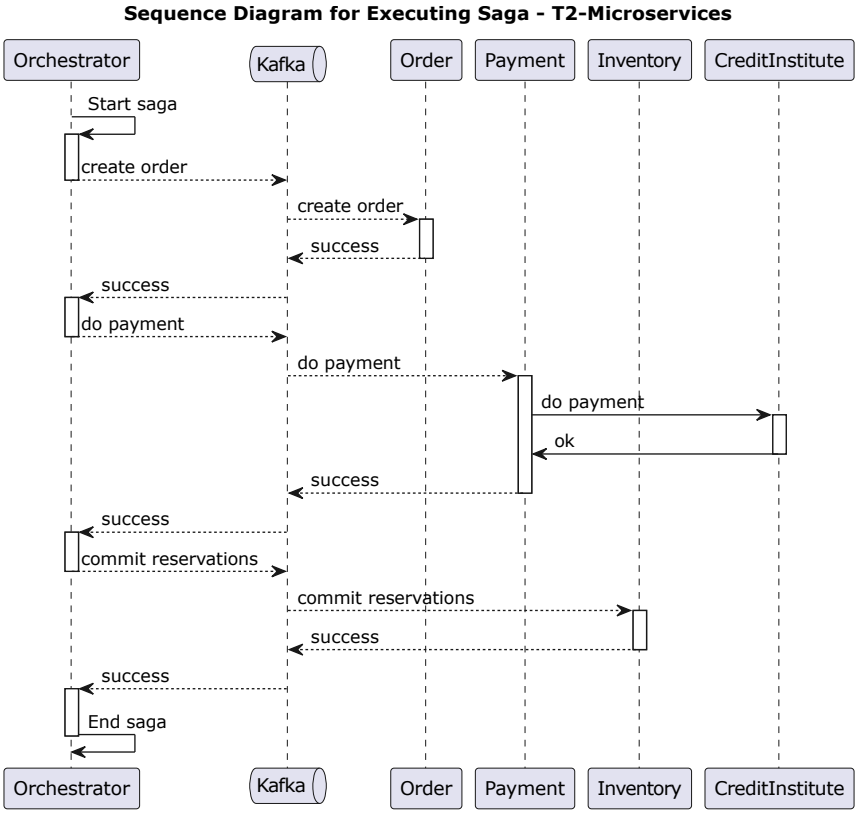

Figure 4.8: Sequence diagram for executing saga using the T2-Microservices variant

4.7 Deployment View

The T2-Project is designed as a cloud-native project. The deployment of the individual components is container-based accordingly and is designed for orchestration with *Kubernetes*.[17] For development or testing, the T2-Project can also be started quickly and easily using *Docker Compose*.[18] The required configuration files are located in the GitHub project "t2-project/devops".[19]

Figure 4.9 shows the deployment diagram for the T2-Modulith variant and Docker Compose. The Modulith is deployed as a `war` file to *Apache Tomcat*,[20] a Jakarta EE server, which is used by default with Spring MVC. `war` is used instead of `jar` because JSP, which is required for the UI, only works with `war`. The latest MongoDB version 7.0 is used for the MongoDB instance. However, the PostgreSQL version 12 used is not the very latest and is only used to enable better comparability with the microservices variant. The PostgreSQL instance of Eventuate used there still uses version 12. This version is supported until November 2024.[21]

[17] https://kubernetes.io
[18] https://docs.docker.com/compose
[19] https://github.com/t2-project/devops
[20] https://tomcat.apache.org
[21] https://www.postgresql.org/support/versioning

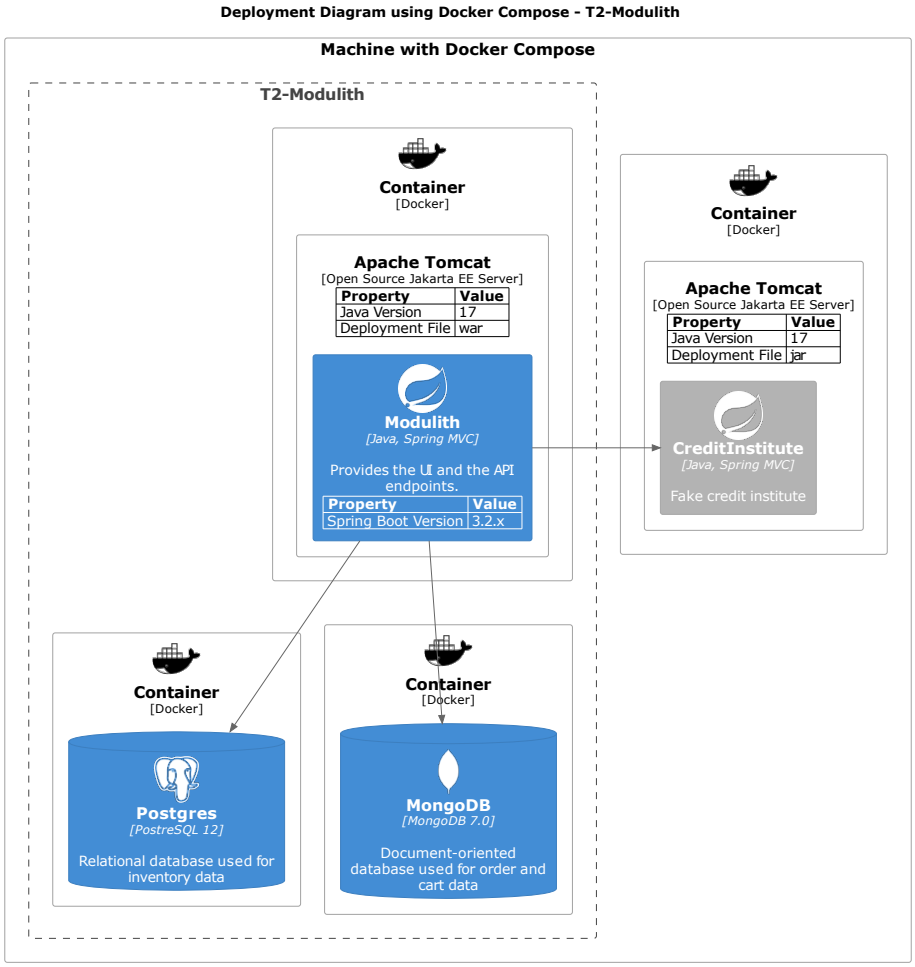

Figure 4.9: C4 deployment diagram using Docker Compose and the T2-Modulith variant

A deployment diagram of the microservices variant and Docker Compose is not provided here as it does not add any additional value. The composition of the microservices has already been described in the Section 4.5 “Building Block View” and the information on deployment is easily comparable with the Modulith variant. In terms of the deployment file, the UI service is also deployed as a `war`, while all others are deployed as `jars`.

The deployment of the T2-Modulith with Kubernetes using the Elastic Kubernetes Service (EKS)[22] in the AWS cloud is shown in Figure 4.10. The diagram is kept very simple in order to focus on the most important aspects. The modulith and its databases are operated on a worker node of EKS, for which AWS EC2 instances[23] are used. In contrast to the local Docker Compose setup, the modulith is not directly accessible, but only via a network load balancer[24] provided by AWS. This also provides a publicly accessible Uniform Resource Locator (URL).

[22] https://aws.amazon.com/eks
[23] https://aws.amazon.com/ec2
[24] https://aws.amazon.com/elasticloadbalancing/network-load-balancer

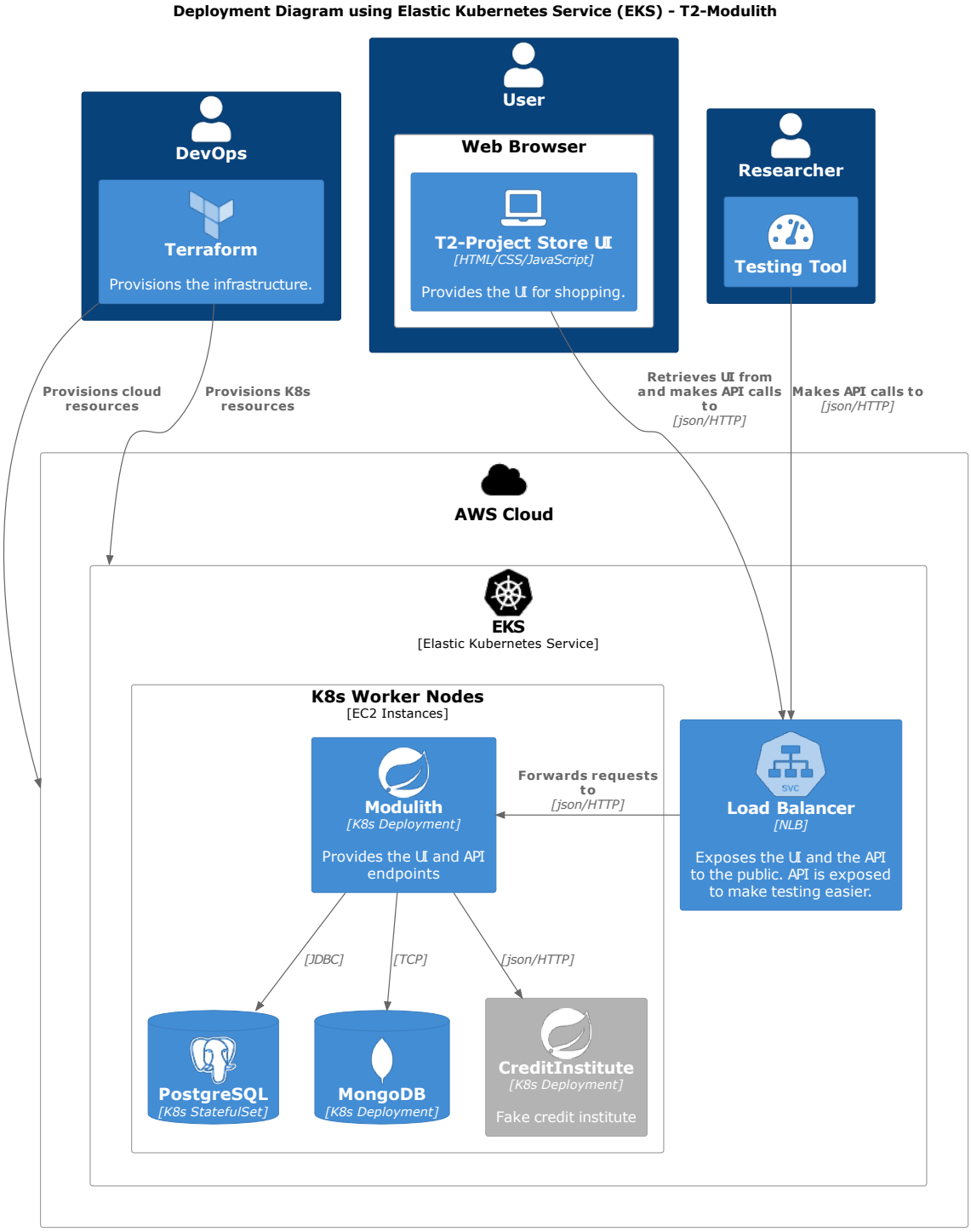

Figure 4.10: C4 deployment diagram using the Elastic Kubernetes Service and the T2-Modulith variant

In addition, Figure 4.10 shows three roles to better illustrate the external interfaces of the system:

- *DevOps:* Provides the required infrastructure resources and Kubernetes resources with the help of Terraform.[25]
- *User:* Accesses the UI with a web browser using the public URL provided by the network load balancer.
- *Researcher:* The public interface provided by the load balancer can also be accessed to carry out tests. The API of the modulith is therefore also exposed to the public in order to simplify tests. This does not correspond to the approach in a real environment. Port forwarding using `kubectl port-forward` is not suitable for many tests, as this could affect the results and for example no load balancing is performed between several existing pods.

The deployment of the T2-Microservices variant with EKS is shown in Figure 4.11 in a simplified form. Relevant differences between the deployments of the monolithic and the microservices variants are that there are two load balancers for the microservices, as the UI and UIBackend each require their own load balancer, and there are now more middleware components: Kafka, Zookeeper and one additional PostgreSQL and MongoDB database (the additional databases are shown in the diagram to save space).

Later, in the Chapter 6 “Experiment Setup & Execution”, the deployment diagrams are expanded with the components required to perform various measurements.

4.8 Cross-Cutting Concepts

This section deals with some important concepts that are relevant for several components of the system and should therefore be discussed explicitly here.

4.8.1 Modularity

When creating the T2-Modulith, it was essential to create the architecture based on bounded contexts in the sense of DDD and thus maintain the modular structure of the microservices. With a monolith, however, it can quickly happen that module boundaries are violated. It is therefore very helpful if the previously defined module limits are validated with the help of tooling.

With T2-Modulith, Spring Modulith is used in particular to validate the module boundaries. If a violation occurs, a unit test fails so that the entire build in the CI pipeline also fails. Compliance with the module boundaries is therefore always ensured. Spring Modulith also generates a component diagram each time the unit tests are run, which visualizes the module structure. This diagram in a slightly adapted version has already been shown in the section 4.5.1 Level 1 of the Building Block View (Figure 4.5).

[25] https://www.terraform.io

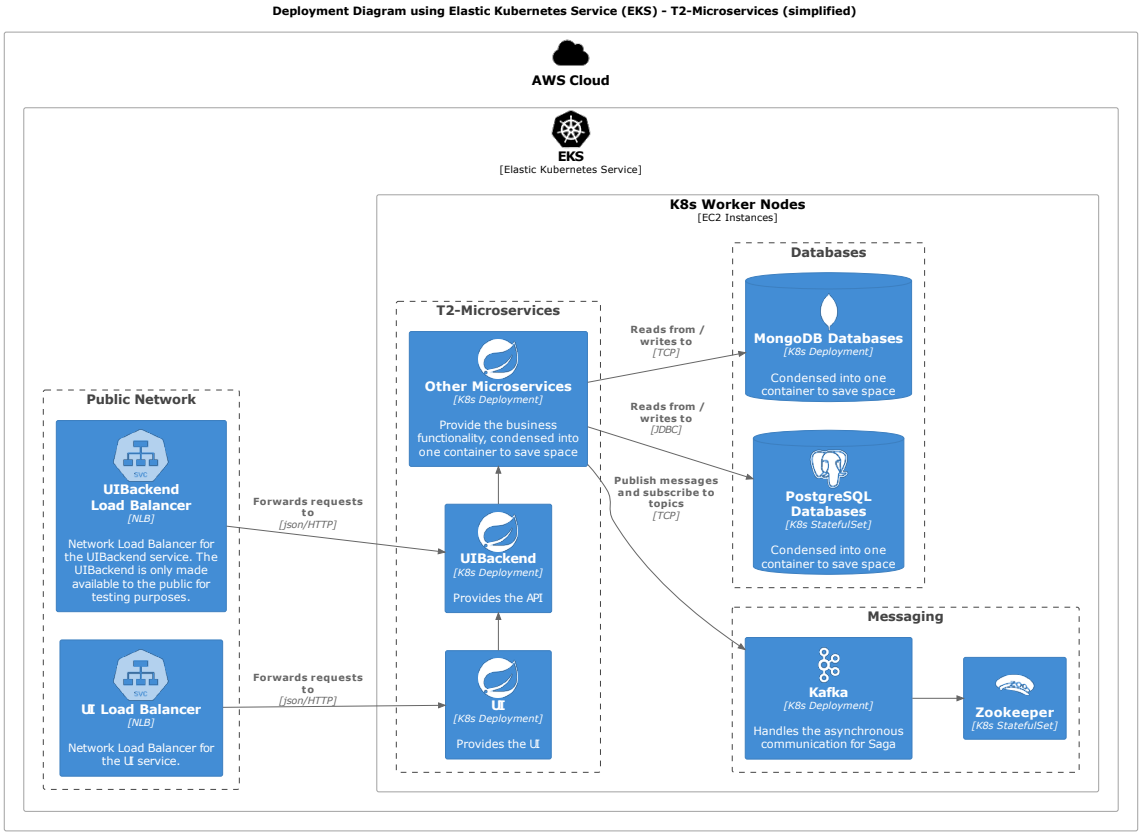

Figure 4.11: C4 deployment diagram using the Elastic Kubernetes Service and the T2-Microservices variant

Spring Modulith offers many more functionalities, but these have not yet been used in T2-Modulith. In particular, the Spring Modulith documentation[26] recommends the use of application events for communication between modules. This is recommended in order to better decouple modules from each other, as a module then no longer needs to know which other modules exist. During the creation of the T2-Modulith, the decision was made not to implement this, as the additional effort of restructuring would not have provided any added value in the context of this thesis.

4.8.2 Monitoring

Monitoring is an essential aspect of cloud-native applications. The T2-Project uses the popular monitoring stack based on Prometheus[27] and Grafana.[28] The monitoring setup for the T2-Project is shown in Figure 4.12.

[26] https://docs.spring.io/spring-modulith/reference/events.html
[27] https://prometheus.io
[28] https://grafana.com

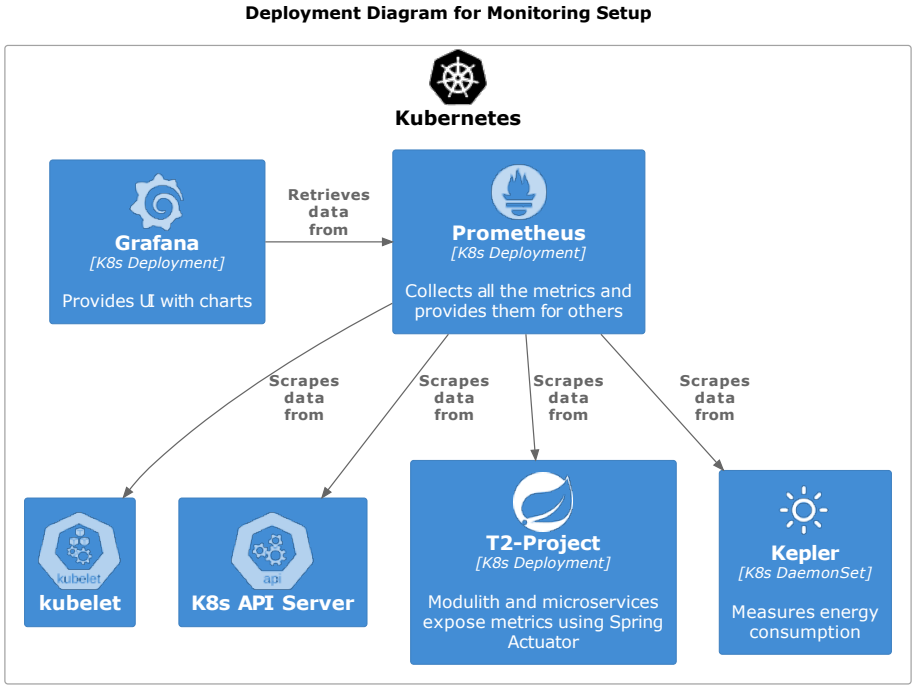

Figure 4.12: C4 deployment diagram showing the monitoring setup of the T2-Project

Prometheus scrapes metrics from various sources and makes them available to others via a standardized interface. Grafana is used to graphically present the metrics provided in charts. The sources used for the metrics include the kublets on the individual Kubernetes nodes, the Kubernetes API server and custom deployments that provide an standardized API for Prometheus metrics. The modulith and microservices provide metrics for Prometheus using Spring Boot Actuator.[29] Kepler provides metrics on energy consumption that can be visualized in Grafana. More information about the setup with Kepler is provided in Section 6.3.

4.8.3 Autoscaling

Autoscaling is also a fundamental component of cloud-native applications. In particular, the horizontal scaling of pods in Kubernetes is an important functionality in order to be able to react well to increased load (scale-out) on the one hand and to be able to save resources with reduced load (scale-in) on the other hand. In the T2-Project, the horizontal scaling of pods is realized using the default Kubernetes *Horizontal Pod Autoscaler (HPA)*.[30] The HPA can scale pods based on CPU and/or memory utilization. It queries the metrics required for this via the Kubernetes metrics API. The T2-Project uses the *Prometheus Adapter for Kubernetes Metrics APIs*[31] to provide the required metrics API. The autoscaling setup is shown in Figure 4.13.

[29] https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#actuator.metrics.export.prometheus
[30] https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale
[31] https://github.com/kubernetes-sigs/prometheus-adapter

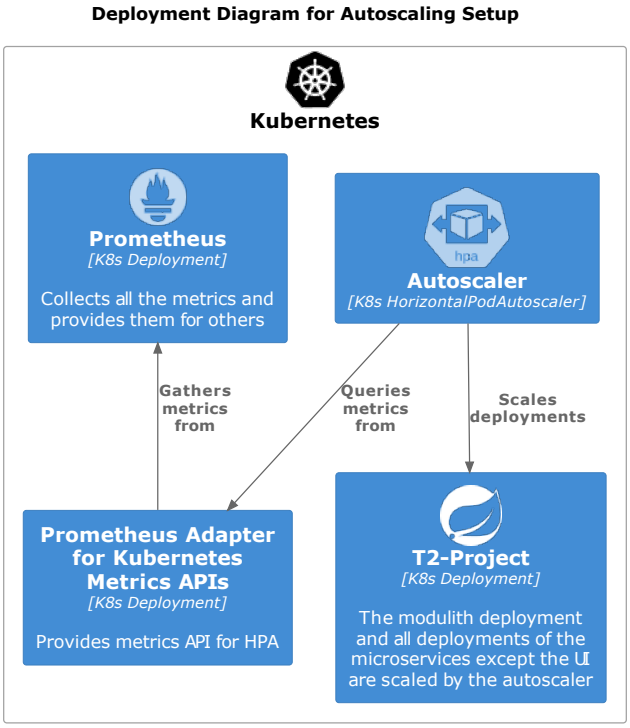

Figure 4.13: C4 deployment diagram showing the autoscaling setup of the T2-Project

In addition to automatic scaling at pod level, automatic scaling at cluster level would also be advantageous and important in order to save resources. With cluster scaling, for example, the cluster is scaled depending on the requested number of pods so that a sufficient number of nodes are available, possibly also in an individually suitable size, and scale-in also takes place if necessary in order to save resources. However, cluster scaling has not yet been implemented for the T2-Project.

4.8.4 Data Persistency & Data Consistency

This section deals with the technologies used for data persistence on the one hand and the aspect of data consistency on the other. The two topics are examined together in this section, as they are interdependent.

Database Technologies

Two different database technologies are used for data persistence in the T2-Project: PostgreSQL as a relational database and MongoDB as a document-oriented database. PostgreSQL is used for inventory data and saga data, while MongoDB is used for data from the order and cart service. This decision was made during the implementation of the initial version of the T2-Project. The specific reasons for this are unknown. When creating the modulith, this was adopted in order to ensure comparability.

Database per Service

With microservices architectures, it is common for each microservice to have its own database. This was also initially planned for the T2-Project, but was never implemented. Instead, a MongoDB database instance was used by both the order and the cart service and the PostgreSQL database instance used for saga was also used by the inventory service to store its domain data. However, there was already an open GitHub issue that addressed the problem with the database of the Inventory Service ("fix database of inventory"[32]), as well as an independent pull request with the attempt to implement the database per service pattern ("added additional databases to be in line with architecture"[33]). The issue proposes using two separate databases for the inventory service, one for the transactional outbox pattern and another for the domain data. However, this contradicts the transactional outbox pattern, which requires that both the domain data and the required outbox table are located in the same database in order to enable the atomicity of local transactions.

The actual solution for implementing the transactional outbox pattern is to provide a separate ACID-capable database for each service that participates in the saga and requires a database, and to give the Eventuate CDC service access to this database. This is easy to implement for the inventory service. An additional PostgreSQL database simply needs to be created and the database access configurations of the inventory service and CDC service adjusted accordingly.

The order and payment services also participate in the saga. The payment service does not require a database. The order service uses a MongoDB database to store its domain data. The problem that the MongoDB database is currently used by both the order service and the cart service can be easily solved: duplicate the MongoDB database and adjust the configurations of the services. However, ensuring transactional integrity cannot be guaranteed, as MongoDB is not supported by the saga framework Eventuate Tram. This aspect is dealt with in more detail in the following section.

Transactional Integrity

Ensuring data consistency is one of the fundamental goals of the T2-Project. How this is ensured in the T2-Microservices variant with the help of saga and the transactional outbox pattern has already been explained in the Section 4.6 on the Runtime View. However, as mentioned in the previous section, there is the problem that MongoDB cannot be used to implement the transactional outbox pattern. This aspect is unattractive and contradicts the goal of always guaranteeing transactional integrity. However, as this aspect has no influence on the actual topic of this thesis, the decision was made not to make any further adjustments in this regard.

With the T2-Modulith variant, it is obvious to rely on local transactions. This is also the case with the operations of the inventory module, where local ACID transactions can be carried out with the connected PostgreSQL database. However, the use of MongoDB also poses challenges for the T2-Modulith. For the most important operation, the completion of an order process, the monolith must access both the PostgreSQL database and the MongoDB database. A transaction would therefore have to include both databases. This would theoretically be possible if both databases

[32] https://github.com/t2-project/inventory/issues/2
[33] https://github.com/t2-project/devops/pull/2

supported two-phase commit, but this is not the case with MongoDB. The current implementation therefore does not guarantee transactional integrity here either. But here too, this aspect does not play a decisive role for the actual topic of this thesis, so that no further adjustments are made.

4.8.5 Communication and Complexity

The communication between the services respectively modules differs fundamentally between the two variants of the T2-Project:

- T2-Modulith: Interprocess communication between the modules
- T2-Microservices: Network communication between the services, either synchronously via HTTP or asynchronously for saga using Kafka as a message broker

Network communication involves the need for data processing on both sides for serialization and deserialization of the data. JavaScript Object Notation (JSON) is used as the format for the payload. This type of data processing is completely unnecessary with the Modulith, as the data can be exchanged directly between the modules. This also means that significantly fewer lines of code are required with the modulith.

The following command is used to find out how many Lines of Code (LoC) are required for the microservices and how many for the monolith:

```
  find ./*/src/main/ -type f -name "*.java" -not \( -path "*/scaling*" -o -path "*/gmt*" -o -
path "*/e2etest*" -o -path "*/creditinstitute*" -o -path "*/computation*" \) | xargs wc -l
```

The result is:

- T2-Microservices: 3852 LoC
- T2-Modulith: 2649 LoC

The modulith therefore has 31 % fewer lines of code than the microservices in total. There are two main reasons for this:

- No network communication and thus elimination of the logic for serialization, deserialization and error handling.
- No saga pattern and thus elimination of the orchestrator service (302 LoC).

Although lines of code are not a direct metric for the complexity of software, it can at least be concluded that the elimination of the saga pattern and the logic required for network communication has a positive effect on readability and maintainability.

4.9 Architecture Decisions

This section discusses a few, but important, architectural decisions that were made in the course of this thesis when creating the reference applications and that have not yet been discussed.

4.9.1 Infrastructure Provisioning

Originally, there was no easy way to create the required infrastructure in the cloud for the deployment of the T2-Project. There were only Kubernetes manifest files, which meant that the actual Kubernetes infrastructure had to be created manually. However, the required infrastructure should be able to be created quickly and reproducibly.

The decision was initially made to use EKS[34] from AWS as the primary environment. Alternatively, Azure Kubernetes Service[35] or the Google Kubernetes Engine[36] would also have been conceivable. The decision in favor of the AWS Cloud was ultimately made because the cooperation partner envite consulting GmbH already had a great deal of experience with the AWS Cloud and could provide access.

There are numerous methods and technologies for providing infrastructure resources for the Kubernetes environment in the cloud and other required resources. First, eksctl,[37] the official Command Line Interface (CLI) for EKS, was tried out. This is a CLI that can create and provide infrastructure resources relatively easily using commands. However, a major disadvantage is that adjustments to resources already provisioned can only be made with eksctl in a cumbersome manner. The decision was therefore made to switch to another tool, Terraform.

Terraform[38] is also a tool for provisioning resources in the cloud, but uses a declarative approach, also known as *infrastructure as code*. For provisioning, Terraform compares the existing resources with the new desired configuration and makes the corresponding changes. This means that changes are much easier to implement and can also be versioned thanks to the infrastructure as code approach.

The configuration for providing the infrastructure for the T2-Project can be found in the devops repository in the subfolder terraform.[39] In addition to the AWS Cloud, configurations for Azure and kind[40] are also provided, but in a reduced form.

Great care has been taken to keep the Terraform configuration modular so that it, like the T2-Project itself, can be easily understood, adapted and extended. To separate the supported environments, isolation is used by means of a corresponding file structure. The inspiration for this comes from Yevgeniy Brikman from the company Gruntwork [Bri22].

To make infrastructure provisioning even easier, several scripts are provided for starting and stopping a directly usable environment. For example, the script `aws-start-microservices.sh` ensures that the Kubernetes infrastructure is provided in the AWS cloud, the Kubernetes resources for the T2-Microservices system are created and finally the UI is made accessible from outside with a public URL.

[34] https://aws.amazon.com/eks
[35] https://azure.microsoft.com/en-us/products/kubernetes-service
[36] https://cloud.google.com/kubernetes-engine
[37] https://eksctl.io
[38] https://www.terraform.io
[39] https://github.com/t2-project/devops/tree/main/terraform
[40] https://kind.sigs.k8s.io

4.9.2 UI Integration

When creating the T2-Modulith, the question arose as to whether the UI component should also be integrated into the modulith or whether it should remain a separate service. In practice, with a modern tech stack for both the backend and the frontend, the two parts would definitely be separated. However, the UI currently used by the T2-Microservices system uses a fairly old tech stack based on JSP, provides its own API for the frontend using Spring MVC and communicates with the UI backend microservices via HTTP. An issue[41] to create a new UI for the T2-Project with modern frontend technologies has existed for around two years, but has never been implemented.

Due to how the UI is currently designed, the decision was made to integrate it into the modulith. This also eliminates the need for network communication between the UI component and the UI backend component, as is the case with the T2-Microservices system.

The only limitation of integrating the UI into the modulith is that the modulith must now be provided as a war file instead of a jar. According to the Spring documentation,[42] JSPs are not supported for executable jars. However, there are no known disadvantages to deploying the whole modulith as a war file.

If a new UI is to be created for the T2-Project in the future, it is very easy to remove the UI component from the modulith. Thanks to the modular structure, it is possible to simply remove the UI module without having to make any further adjustments to the source code.

[41] https://github.com/t2-project/ui/issues/1

[42] https://docs.spring.io/spring-boot/docs/current/reference/htmlsingle/#web.servlet.embedded-container.jsp-limitations

4.10 Risks and Technical Debts

The T2-Project has a few architectural shortcomings, which are briefly explained in Table 4.3.

Table 4.3: T2-Project risks and technical debts.

Risk	Description
No transactional integrity	As described in Section 4.8.4 “Data Persistency & Data Consistency”, there are problems with both variants of the T2-Project in terms of transactional integrity. In the case of microservices, this is due to the fact that the transactional outbox pattern cannot be implemented with MongoDB and in the case of the modulith due to the fact that no local transaction can be executed that spans both PostgreSQL and MongoDB. The simplest solution for the latter would be to simply store everything in the PostgreSQL database. This realization came too late for this thesis, but should be evaluated in future work.
Autoscaling of pods is limited to CPU utilization	Pod autoscaling is used with the Kubernetes Horizontal Pod Autoscaler, which is limited to the metric of CPU and memory utilization. It would be better to use KEDA,[43] which would also allow scaling based on the number of messages in Kafka’s queue, for example.
Cluster autoscaling is missing	The lack of cluster autoscaling means that a large number of nodes must be provided in advance, which can also cover peak loads. At the same time, this means that the resources of all nodes are usually underutilized, which is a waste of costs and bad for the environment.
Database operations are not optimized	There is quite a bit of potential for optimization in the database queries. For example, Hibernate generates more operations than necessary and indexes have not yet been used.
Eventuate Tram Framework as a dependency	The microservices have the Eventuate Tram Framework as a dependency, which is a risk. Maintenance and support do not appear to be particularly good. This is shown, for example, by the fact that a small pull request[44] created during the thesis has been unprocessed for over three months. The versions of the Kafka and PostgreSQL instances supplied and used have also not been updated for some time. The official support period for the versions used ends for both later this year.

[43] https://keda.sh
[44] https://github.com/eventuate-foundation/eventuate-common/pull/135

5 Measurement Methodology

As described in the Section 2.6 “Assessing the Energy Efficiency of Software” in the foundations chapter, there are various ways of measuring or estimating the energy consumption of software. This chapter explains the methodologies chosen for this thesis in order to compare the energy efficiency of the modular monolith and microservices architecture variants.

5.1 Preliminary Considerations

With the help of the Green Software Measurement Model (GSMM), described in the Section 2.6.4 in the foundations chapter, the key aspects can be defined and fundamental decisions made. The individual components of the GSMM and the corresponding decisions are described below.

Measured object and measurement goals

The measured object is an implementation of the T2-Project. The goal is to compare two different architecture variants under various conditions.

Measurements and metrics

An *order* is considered to be a good metric for *useful work* in regards to energy efficiency. Therefore, the energy consumption of a simple ordering process consisting of three requests is used as a metric for comparison. The three requests are *get inventory*, *add item to cart* and *confirm order*.

Measurement procedure models

A black box measurement approach is used. Usage scenarios are used to make comparisons based on meaningful interactions with the software. A workload generator is required to execute the usage scenarios. JMeter is selected as the workload generator. For more information, see the next Section 5.2 “Workload Generator”.

Measurement setup

This thesis intends, as far as practically possible, not only to estimate energy consumption, but to actually measure it. Measurement tools that use RAPL seem to represent a good compromise between accuracy and practicality, so that appropriate tools were chosen for this thesis. The Green Metrics Tool from Green Coding Solutions, which is also presented in the paper [GBC+24] on the GSMM, is particularly suitable. Compared to other similar tools, it offers the following advantages, among others:

- Simple container-based setup (the only requirement is a publicly accessible usage_scenario.yml file based on Docker Compose)
- A directly usable measurement cluster is available so that no separate hardware setup for measurements needs to be set up and calibrated
- Reproducibility through defined usage scenarios
- Uses standards such as the SCI specification
- Is actively developed further

As the Related Work chapter has shown, it is essential to consider and measure scaling behavior when comparing the architecture styles monolith and microservices. However, scaling behavior cannot currently be mapped with the Green Metrics Tool, meaning that an additional measurement setup is required in the cloud. For energy measurements in the public cloud, there is currently one tool in particular that is generally used for this purpose: Kepler.

In summary, this results in the following two measurement setups:

1. Green Metrics Tool: Ready-to-use software stack that can be used immediately to measure and record energy consumption. The measurement cluster provided by Green Coding Solutions is used as the infrastructure. Further information on this follows in the Section 5.3 “Measurement of Usage Scenarios”.
2. Kepler: Is operated together with the T2-Project in a Kubernetes cluster in order to be able to measure energy consumption during scaling behavior. Further information on this follows in the Section 5.4 “Measurement of Scaling Behaviors”.

Data evaluation models

As the Green Metrics Tool already comes with a comprehensive dashboard with helpful information and visualizations, this is sufficient for analyzing the results. The setup with Kepler includes a Grafana dashboard that can be used to analyze the results.

5.2 Workload Generator

To carry out reproducible energy measurements, a tool is required that can automatically execute predefined test plans. For load tests and performance tests, there are already numerous tools available for this purpose, such as *Apache JMeter*,[1] *k6*,[2] *Gatling*,[3] *wrk*,[4] *Artillery*,[5] *Hyperfoil*,[6] *Siege*[7] and *Chauffeur WDK*.[8] For this thesis, JMeter was chosen as it is the most widely used of all the tools mentioned and provides all the functions required.

JMeter enables the creation of extensive test plans using a graphical user interface. Within the test plans, users can use a variety of samplers to send different types of requests to the target application. This includes HTTP requests, Java Database Connectivity (JDBC) database queries, File Transfer Protocol (FTP) requests and more. A central aspect of test plan execution in JMeter is the use of thread groups. These define the behavior of concurrent user requests. The actual execution of the test plans can be done either via the graphical user interface of JMeter or via the command line. For load tests, JMeter should always be used in the command line mode. During test execution, assertions can be used to ensure that the responses of the target application match the expected results. Finally, JMeter offers various listeners that can be used to monitor and analyze results. These include metrics such as throughput, response times, error rates and more to evaluate the performance of the target application. JMeter test plans are stored in the so-called "Java Management Extensions (JMX)" (Java Management Extensions) format. JMX is an Extensible Markup Language (XML)-based format that contains all configuration settings, samplers, assertions, thread groups and other elements of the test plan. When executing test plans, it is possible to include parameters and environment variables. This makes it possible to create flexible and adaptable test plans.

5.3 Measurement of Usage Scenarios

The Green Metrics Tool (GMT) is the perfect tool for measuring usage scenarios. A special characteristic of the GMT is the use of a *usage_scenario.yml* file. This describes on the one hand how the software system to be measured can be provided with the help of containers and on the other hand how the usage scenario is defined for which the measurement is to be carried out. The format of *usage_scenario.yml* is based on a subset of the Docker Compose specification,[9] with a few additional options. The advantage of using the specification is that it is easy for many modern applications to use GMT without much effort. Many already have a Docker Compose file defined or it is easy to create one. The team behind GMT is also convinced that mapping typical usage

[1] https://jmeter.apache.org
[2] https://k6.io
[3] https://gatling.io
[4] https://github.com/wg/wrk
[5] https://www.artillery.io
[6] https://hyperfoil.io
[7] https://www.joedog.org/siege-home
[8] https://www.spec.org/chauffeur-wdk
[9] https://docs.docker.com/compose/compose-file

scenarios is the best way to measure energy [Sol22]. The concept of standard usage scenarios has already been explained in Section 2.5.8. For this thesis, commands are executed in the flow part that send HTTP requests to the API of the T2-Modulith or T2-Microservices system.

The philosophy of the GMT is that a usage scenario should contain all components that reflect the actual use case of the software. This means that if a client is involved, it should be included in the measurement. This differs significantly from other tools that attempt to isolate the energy consumption of individual components (see Section 2.6.2 “Process Allocation” in the foundations chapter). Details on this can be found in a corresponding discussion with Arne Tarara, the main developer of the Green Metrics Tool, [Tar23a]. In addition to looking at the Green Metrics Tool documentation, it is generally recommended to look also at the discussion contributions[10] in the Green Metrics Tool GitHub repository.

GMT uses a variety of different metrics to measure energy consumption, which are integrated with the help of small metric reporters. These include “Memory per Container”, “CPU % per Container”, “CPU Time per Container”, “Network Traffic per Container”, “CPU Frequency”, “CPU Temperature”, “Fan speed”, “DC Energy of System” and “AC Energy of System”. The use of the individual metrics is configurable, so that execution is also possible on systems on which only some metrics can be recorded.

Green Coding Solutions believes that transparency and reproducibility are important factors in promoting awareness of software energy consumption [Sol22]. Accordingly, the Green Metrics Tool is provided as Free and Open Source Software (FOSS) and, by providing a measurement cluster and the public Green Metrics Frontends,[11] enables measurements to be tracked and comparisons to be made on prepared, permanently installed machines.

The measurement cluster comprises several machines with different hardware, which are documented in the GMT documentation[12] and from which a suitable machine can be selected. The machines are specially prepared for energy measurements and regular calibration ensures that there are no significant fluctuations between individual measurement runs. The measurement cluster can be used free of charge for one run per day and per repository. Premium access is required for more measurements. For this thesis, permission was kindly granted to carry out more measurements free of charge.

5.4 Measurement of Scaling Behaviors

Kepler is used in a Kubernetes cluster with autoscaling enabled to measure the energy consumption of workloads that scale dynamically on demand. Kepler can measure or estimate the energy consumption of all containers running in the cluster. If low-level interfaces such as RAPL and others are available, Kepler can use them to record the energy consumption. In the public cloud, this is only possible on bare metal instances. In virtualized environments, Kepler uses a model-based

[10] https://github.com/green-coding-solutions/green-metrics-tool/discussions
[11] https://metrics.green-coding.io
[12] https://docs.green-coding.io/docs/measuring/measurement-cluster

approach to estimate energy consumption using a model and usage metrics. There are already predefined models that have been trained on different hardware and can be used directly. If required, custom models can also be trained.

Kepler follows a different philosophy than the Green Metrics Tool and attempts to determine the energy consumption for each container at a fine-grained level. It uses the number of CPU instructions that a process requires to assign a share of the total energy consumption to a process.

Kepler provides the collected data for Prometheus, which can ultimately be used to query the data and typically visualize it with Grafana.

6 Experiment Setup & Execution

How the experiments with the Green Metrics Tool and Kepler were carried out and what lessons were learned in the process is covered in this chapter. The results are then discussed in the Chapter 7 “Results & Discussion”. JMeter is used as a workload generator for all experiments, so its setup is explained first.

6.1 Using JMeter as a Workload Generator

The Section 4.6 “Runtime View” has already shown what the order process looks like from the perspective of a normal user. The execution of a test plan with JMeter is very similar, as shown in Figure 6.1.

There are only two differences in the process due to automation:

1. Generate a session ID at the beginning and use it for subsequent HTTP POST requests.
2. Automatically select a random item after receiving the inventory and use the ID for the subsequent request to add the product to the shopping cart.

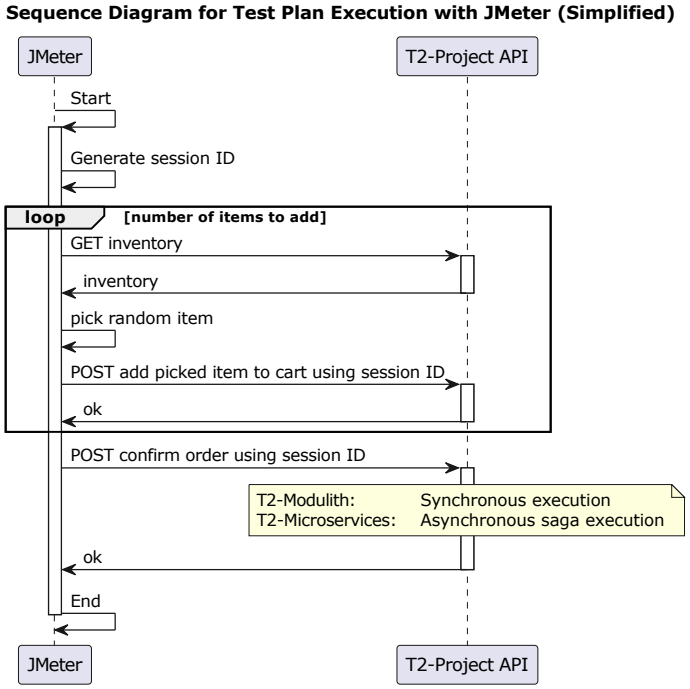

Figure 6.1: Sequence diagram for the test plan execution with JMeter (simplified)

A test plan called "t2-project-flexible.jmx" was created for this purpose, which can process this sequence automatically and can be configured with numerous parameters in the form of environment variables. The basic command for executing the test plan with JMeter without optional parameters looks like this:

```
jmeter -Jhostname=backend -Jport=8080 -n -t t2-project-flexible.jmx
```

This command ensures the execution of three requests as shown in the sequence diagram to the target URL http://backend:8080. The loop is run once, as only one product is added to the shopping cart by default in the test plan. Numerous optional parameters are defined in the test plan, which can be used, for example, to run the loop several times or to execute the requests from several users in parallel. Optional logging is also integrated into the test plan (not shown in the diagram), which can be activated with an environment variable. Logging can be useful for debugging purposes or for calculating the SCI score. The available parameters will not be explained in detail here. A list of all parameters and explanations can be found in the T2-Project user documentation.[1]

The asynchronous execution of saga ensures that JMeter does not notice if something goes wrong during saga execution and cannot record how long it takes for the order process to be completed in full. This is a weakness of the setup, but can be improved in a workaround (explained in the following section).

6.2 Experiments with the Green Metrics Tool

This section describes the setup with the Green Metrics Tool and the adjustments that were necessary to be able to carry out measurements with the GMT. Finally, some of the lessons learned in using the GMT are presented.

6.2.1 Setup

Figure 6.2 shows the measurement setup with the Green Metrics Tool on one of the measurement machines provided, which measures a usage scenario from the T2-Modulith.

The GMT deployment consists of four components:

- *GMT Frontend*: It is accessible via a public URL.[2] Four pages are relevant for the measurement: "Measure software"[3] for submitting new measurement jobs, "Status"[4] for an overview of the available machines, "Home"[5] for an overview of the most recently executed measurements and "Repositories"[6] to view all measurements from specific repositories.

[1] https://t2-documentation.readthedocs.io/en/latest/measurements/jmeter.html
[2] https://metrics.green-coding.io
[3] https://metrics.green-coding.io/request.html
[4] https://metrics.green-coding.io/status.html
[5] https://metrics.green-coding.io/index.html
[6] https://metrics.green-coding.io/repositories.html

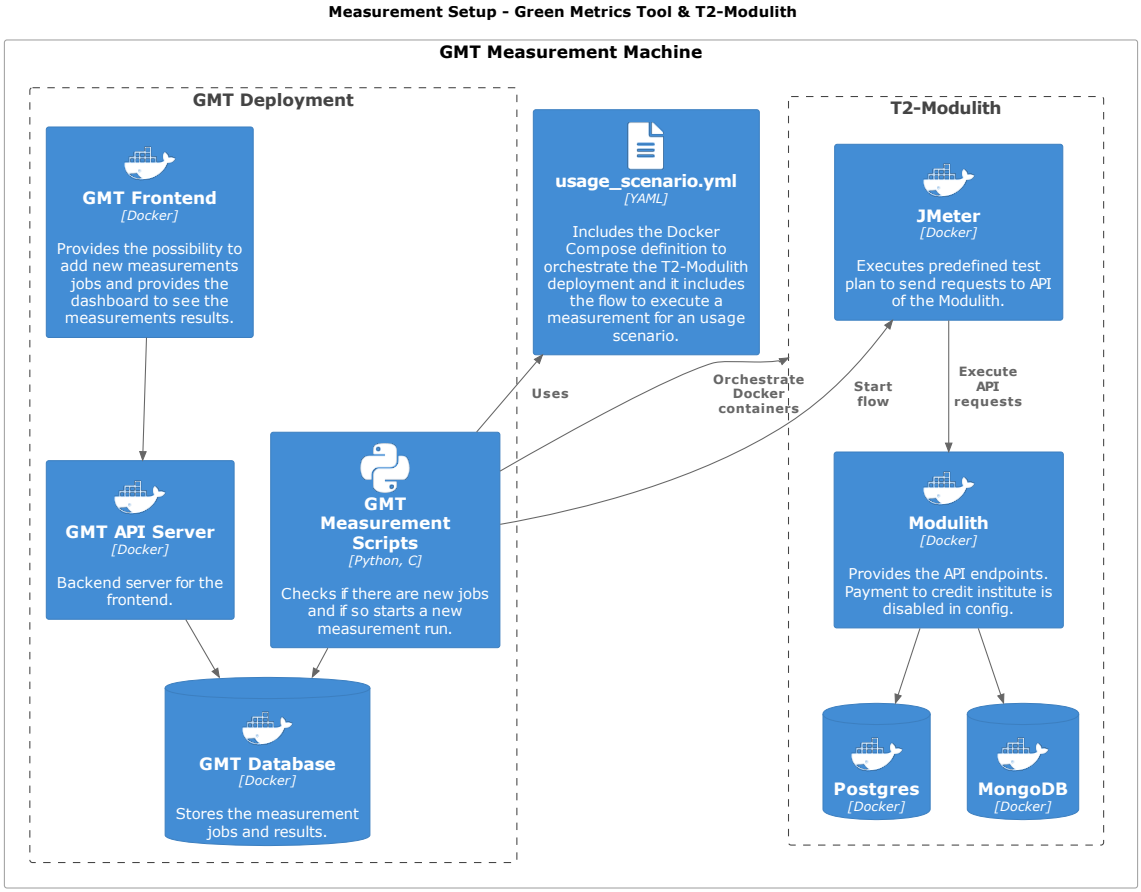

Figure 6.2: C4 deployment diagram showing the measurement setup with the Green Metrics Tool and the T2-Modulith variant

- *GMT API Server*: Backend for the GMT frontend, provides the interface to the database with the measurement jobs and measurement results, among other things.
- *GMT Database*: Database for the measurement jobs and results, among other things.
- *GMT Measurement Scripts*: Python scripts that are used to perform the measurements. If a new measurement job is executed, first the *usage_scenario.yml* gets downloaded, processed and validated. Afterwards the measurement environment is prepared accordingly and the Docker containers for the system to be measured are started. Finally, the measurement is started using the flow provided and the results are saved into the database.

The T2-Modulith is started in Docker containers by the GMT measurements scripts as shown in Figure 6.2 and stopped again after the measurement is complete. Compared to the deployment of the T2-Modulith described in Section 4.7 (Figure 4.9), there is now another container: JMeter. JMeter is used for the automated execution of several HTTP requests to the modulith API in order to use it to represent usage behavior. To avoid having to create a separate test plan for JMeter for each usage scenario, environment variables are used that are passed to JMeter as part of the flow in usage_scenario.yml when it is called.

An example usage_scenario.yml file is shown in Listing 6.1.

Listing 6.1 Example of a basic usage_scenario.yml used by the Green Metrics Tool for energy measurements.

```
name: T2-Modulith Basic Usage Scenario
author: David Kopp
description: One user checks out the inventory, thinks for 30 sec, adds a random product to
cart, thinks again, add a second product, thinks again, add a third product, and finally
confirms the order.

compose-file: !include monolith-compose.yml

sci:
  R_d: order

flow:
  - name: 1 user orders 3 products with 30 sec think time
    container: jmeter
    commands:
    - type: console
      command: jmeter -Jhostname=backend -Jport=8080 -JnumExecutions=1 -JnumUser=1 -JrampUp=0
-JnumProducts=3 -JthinkTimeMin=30000 -JloggingEnabled=true -JloggingSCIEnabled=true
-JloggingSCIEnabled=true -n -t /tmp/repo/t2-project-flexible.jmx
      log-stdout: true
      read-notes-stdout: true
      read-sci-stdout: true
```

With `compose-file` another file is included, in this case the file `monolith-compose.yml`. This is a standard-compliant Docker Compose file that contains the deployment information for the T2-Modulith system and JMeter. `sci` is used to specify how the optional SCI score (described in Section 2.5.7) is to be calculated. In this case, `order` is used as the functional unit. The test plan for JMeter contains logging statements after each `POST confirm order` request, with the message `GMT_SCI_R=1`, which are evaluated by GMT. In the `flow` part, the usage scenario can be modeled in several steps. In this case, one step is sufficient to start the JMeter test plan.

The setup for T2-Microservices is very similar to that for the T2-Modulith, so that no explanation is given here.

6.2.2 Execution

The measurement cluster from Green Coding Solutions was used to carry out relevant measurements with the Green Metrics Tool. During the development and testing of new usage scenarios, GMT was also used locally. The usage scenarios for execution with the Green Metrics Tool can be found in the public GitHub repository "t2-project/devops"[7] in the subfolder "energy-tests/gmt".

[7] https://github.com/t2-project/devops

New measurement runs can be created via the "Measure software"[8] page in the GMT frontend. In a scenario with the name "T2-Modulith (Minimal Scenario with JMeter)", for example, this looks as follows:

- *Name:* T2-Modulith (Minimal Scenario with JMeter)
- *URL:* `https://github.com/t2-project/devops`
- *Filename:* `energy-tests/gmt/monolith-usage_scenario-minimal-base.yml`
- *Branch:* `main`
- *Hardware:* Fujitsu Esprimo P956
- *Measurement:* One-Off [Free - Fair use]

All performed measurements runs can be viewed via the GMT frontend on the "Repositories"[9] page and compared with each other if required.

6.2.3 Required Adjustments

A few adjustments had to be made to the T2-Project, the JMeter Container image used and the Green Metrics Tool in order to be able to carry out measurements successfully. The necessary adjustments are briefly described in this section.

Adjustments to the T2-Project

The T2-Project uses a fake service as a credit institute for payments. Due to the original orientation of the T2-Project, the CreditInstitute service is designed to randomly provoke SLO violations. This is not acceptable for reproducible energy measurements. Furthermore, there is no added value in including this service in the energy measurement. The decision was therefore made to omit the service completely and to make the call to the CreditInstitute service optional via configuration in the payment service. This means that no CreditInstitute service is required for measurements with GMT and the payment service (T2-Microservices) or the payment module (T2-Modulith) returns an ok directly without requesting the CreditInstitute service.

For the deployment of the T2-Microservices system, a change had to be made to the container image for the PostgreSQL databases, which is provided by Eventuate. The image cannot be executed automatically in the interactive mode of Docker (parameter `-it`), as is done by GMT. A corresponding pull request[10] was created, but not merged in the period of this thesis. For this reason, a self-created container image[11] is used, which contains the required change.

[8] https://metrics.green-coding.io/request.html
[9] https://metrics.green-coding.io/repositories.html
[10] https://github.com/eventuate-foundation/eventuate-common/pull/135
[11] https://hub.docker.com/r/t2project/eventuate-tram-sagas-postgres

The Orchestrator microservice has been extended with an optional logging mechanism[12] to be able to log the end of the asynchronous saga process with the message `GMT_SCI_R=1` and the corresponding timestamp. This is relevant so that GMT can calculate the SCI score correctly, taking into account not only the synchronous part of the order process, but also the asynchronous part.

Adjustments to the JMeter Container Image

For JMeter, the container image "justb4/jmeter"[13] is used, which is the most downloaded image for JMeter at DockerHub. It is designed so that JMeter is started using `docker run` and the container terminates as soon as JMeter has finished executing a test plan. This is unsuitable for the use with GMT. A container must always be running in a GMT setup and must be able to execute commands at runtime using `docker exec`. This requires the entry point of the container image to be adapted accordingly so that JMeter is not started immediately when the container is started. The customized version can be found in the GitHub repository "t2-project/docker-jmeter"[14] and at DockerHub under the name "t2project/jmeter".[15]

Adjustments to the Green Metrics Tool

When using the Green Metrics Tool with the T2-Project, there were a few problems that led to the creation of some pull requests:

- #541 "Allow digits in env var keys"[16] – digits are used in the keys of environment variables
- #571 "Make Docker image tags lowercase"[17] – the container image for PostreSQL uses a tag name with capital letters, which caused problems (`0.23.0.RELEASE`)
- #577 "Refactor env var checks and tests"[18] – the T2-Project uses environment variables with characters that were forbidden by GMT
- #584 "Fix frontend flow menu to wrap automatically"[19] – if more than five flow steps were used, the other steps were not visible in the frontend
- #590 "Support reading notes from service"[20] – SCI score calculation should include the asynchronous saga operations of the T2-Project
- #593 "Use depends_on for container startup order (refactored)"[21] – the T2-Project components have dependencies that must be started in the correct order

[12] https://github.com/t2-project/orchestrator/commit/383febd
[13] https://hub.docker.com/r/t2project/jmeter
[14] https://github.com/t2-project/docker-jmeter
[15] https://hub.docker.com/r/t2project/jmeter
[16] https://github.com/green-coding-solutions/green-metrics-tool/pull/541
[17] https://github.com/green-coding-solutions/green-metrics-tool/pull/571
[18] https://github.com/green-coding-solutions/green-metrics-tool/pull/577
[19] https://github.com/green-coding-solutions/green-metrics-tool/pull/584
[20] https://github.com/green-coding-solutions/green-metrics-tool/pull/590
[21] https://github.com/green-coding-solutions/green-metrics-tool/pull/593

6.2.4 Learnings & Improvements

Many measurements were carried out with the GMT as part of this thesis. Accordingly, there were some learnings that are now presented below.

Idle Energy Consumption With GMT the absolute energy consumption value is not really important, because this value depends on many variables, especially the machine and environment. Therefore, the results are usually only relevant for relative comparisons between different runs. It's important that the energy consumption of the machine in idle mode (baseline) is the same between runs, so it doesn't influence the results. The team behind the GMT ensures this by executing a measurement that should always give the same result regularly: *Measurement Control Workload*.[22]

Energy Overhead by JMeter GMT can only measure the energy consumption of the whole system that is part of an usage scenario. Therefore, the energy consumption of JMeter is always included in the resulting energy values. The measurement of individual components is not possible with GMT, because there is no clear way of how to isolate individual components and GMT has the philosophy that a usage scenario should contain all components to reflect an actual use case of the software. Therefore, all components that are part of an usage scenario are also part of the energy measurement. For comparisons between different applications this should not be a problem, as long as the respective components behave the same. In theory, that should also be the case with JMeter that always executes the same test plan (perhaps with different parameters, so that have to be kept in mind for comparisons). However, measurements with the GMT setup have shown that the start process of JMeter can take different lengths of time (3–10 seconds), so that this can have a negative effect on the results. This must be taken into account when comparing measurement results.

Network Energy Estimation The metric "Network Transmission Energy" that is part of the measurement results shown in the GMT frontend refers to the estimated energy consumption by network traffic in a distributed global system. The value is calculated by the total amount of sent and received bytes from the network interface multiplied by a constant value. The constant used is the one calculated by Aslan et al. down to 2024, i.e. 0.002652 kWh/GB at the time of writing this thesis (see Section 2.5.3).

It is therefore important to note that the value is not the energy consumption of network communication within a machine or data center, but the potential energy consumption that arises when the system is operated in a globally distributed way. This is not the case for a typical microservices system, which is operated in a data center or in several data centers in the same region. The Cloud Carbon Footprint Tool (see Section 2.5.6) does not include such network communication within a data center at all. Furthermore, it is generally questionable how useful it is to estimate the energy consumption of network communication using a constant for GB/kWh, as discussed in Section 2.5.3 "Energy Consumption of Network Data Transfer". The team behind GMT decided to use this methodology "as it bests incentives the user to keep the network traffic to a minimum" [Tar24].

[22] https://metrics.green-coding.io/timeline.html?uri=https://github.com/green-coding-berlin/measurement-control-workload&filename=usage_scenario.yml&branch=event-bound&machine_id=7

Measure asynchronous operations The runtime phase in GMT is based on the defined flow: it starts with the execution of a command and ends when the command is finished. If the command triggers an asynchronous operation the flow/phase may end before the asynchronous operation actually has finished.

So the question arises as to how the entire operation can be recorded and measured? As an workaround a sleep command is used to extend the duration of the flow. This is coupled with the challenge of determining a suitable duration. Together with logs from the orchestrator, a suitable duration can be determined so that the measurement can be repeated with a more suitable duration.

Warm-up of application Applications with a runtime environment and a Just In Time (JIT) compiler optimize themselves during runtime. This is the case with the HotSpot JVM used by the T2-Project services. Therefore, in performance benchmarking, it is common practice ignore the first measurements during *warm-up* and only consider the measurement results when the application is considered to be *warm*.

The question arises as to how this should be handled in energy measurements. Arne Tarara is convinced that the warm-up phase must also be included in the energy measurement [Tar23b]. Tarara argues that these are operations that must be performed at a certain point in time to bring the application to its operating point, thus consuming energy and causing carbon emissions.

However, for a better understanding of the results, it seems useful to have measurement results for the warm-up phase as well as for the actual execution of a scenario and not to mix them together. GMT does not support an explicit warm-up phase. However, due to the flexibility in the definition of a usage scenario, an additional step can easily be defined in the flow part in usage_scenario.yml, which can be used as a warm-up.

Impact of Logging Logging of all requests by JMeter requires a significant amount of energy (in a test scenario e.g. +13 %). Therefore, it should be enabled only if really necessary, e.g. during testing of new usage scenarios. The GMT documentation page about best practices[23] also recommends to turn logging off to avoid overhead.

High Load and Scaling The original idea of GMT is to execute standard usage scenarios and measure the energy consumption of such. Therefore, GMT is not designed for load testing or similar approaches. However, due to the flexible way in which usage scenarios can be defined, it is easily possible to generate larger loads and measure energy consumption. However, it must be taken into account that GMT cannot be used to create dynamic scaling scenarios. When a measurement is started, exactly one container instance is started for each service and no horizontal scaling is possible at runtime.

[23] https://docs.green-coding.io/docs/measuring/best-practices/#13-turn-logging-off

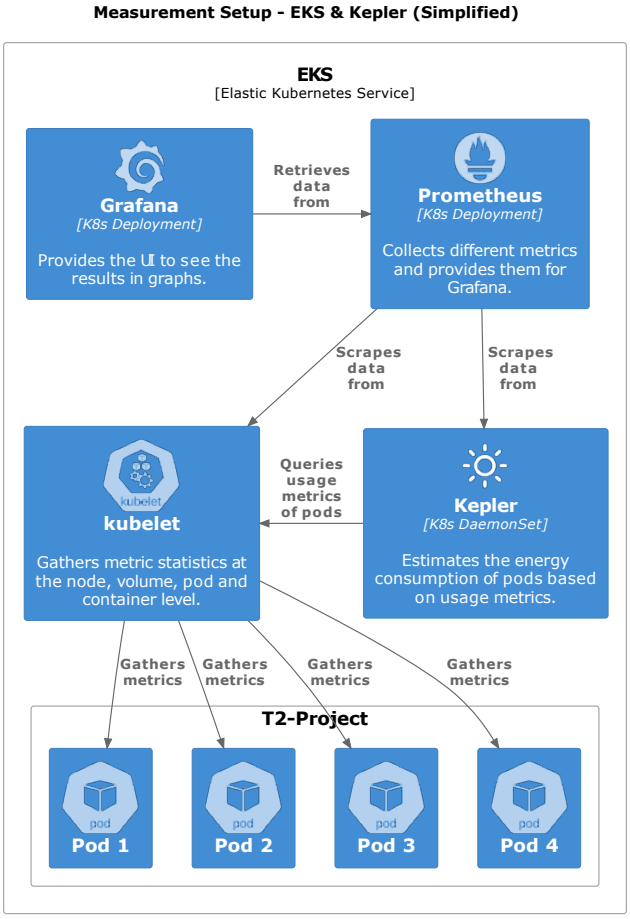

Figure 6.3: C4 deployment diagram showing the measurement setup with EKS & Kepler

6.3 Experiments with Kepler in a K8s Cluster

This section is dedicated to the setup for energy measurements with Kepler.

6.3.1 Setup

The individual components required for the deployment of the T2-Project with Kepler in a Kubernetes environment have already been described in the sections Deployment View, Monitoring and Autoscaling. All these components combined create the environment required for energy measurements with Kepler. In Figure 6.3, the measurement setup with Kepler is shown again in condensed form with the most important components. EKS in the AWS cloud is used as the Kubernetes environment.

Since virtualized instances are used and no bare metal, Kepler cannot measure the energy consumption with RAPL or other hardware interfaces itself, but must estimate it with the help of usage metrics. With Grafana, the results can be presented visually in charts.

Terraform is used to provide the required infrastructure as described in Section 4.7 “Deployment View”. The setup for energy measurements with Kepler differs from the normal setup in the following points:

- Both variants, T2-Modulith and T2-Microservices, are operated in the same Kubernetes cluster in different namespaces. Measurements are only ever carried out with one system at a time, but the presence of both systems in the same cluster makes it easier to carry out different measurements quickly with both systems in sequence.
- The Grafana dashboard is made publicly accessible using an AWS load balancer, as there are often disconnections when using `kubectl port-forward` with Grafana.
- The UIBackend service is made publicly accessible using an AWS load balancer to get reliable load balancing for the API gateway. Local port forwarding with `kubectl port-forward` cannot be used as it does not support load-balancing and only forwards the requests to a pod.
- Autoscaling is set up appropriately for energy measurements.
- A compute-intensive scenario can be activated in which a compute-intensive operation is simulated during the *confirm order* operation. In the microservices system, this is done using an additional microservice called *computation-simulator*,[24] which is called by the UIBackend service during the *confirm order* operation, and in the monolithic system, the optional calculation operation is implemented in a separate module, which is loaded via lazy loading as required.
- As with the setup with GMT, no CreditInstitute service is deployed. The payment is deactivated by configuration in order to be able to carry out reproducible measurements.
- The Kubernetes resource management[25] for individual containers (`requests` and `limits`) is adapted to optimize the utilization of the instances and to ensure better autoscaling.
- The size of the product inventory is increased to avoid too many simultaneous database operations on the same data set under heavy load.

The customizations are located in the GitHub repository “t2-project/devops”[26] in the subfolder “energy-tests/k8s”. A script with the name “aws-start-microservices-and-modulith.sh” is available, which can handle the entire deployment including the customizations. Kustomize[27] (`kubectl -k`) is used for the required modifications to the Kubernetes resources. This makes it possible to make specific adjustments to Kubernetes manifests without having to duplicate them.

[24] https://github.com/t2-project/computation-simulator
[25] https://kubernetes.io/docs/concepts/configuration/manage-resources-containers
[26] https://github.com/t2-project/devops
[27] https://kustomize.io

6.3.2 Execution

The execution of measurements with Kepler differs greatly from those with GMT. By using a usage scenario, GMT has a defined sequence of steps with a start and end point for the measurement. With Kepler, the execution of a scenario is decoupled from the measurement, which makes it more difficult to carry out measurements of individual scenarios.

Kepler automatically records the energy consumption of all containers operated in the same Kubernetes cluster at all times. The test plan is executed manually with JMeter independently of this. The energy consumption can then be tracked in the Grafana dashboard with a slight delay, almost in real time. It is not possible to determine the exact energy consumption for a scenario executed by JMeter. However, it is possible to compare different versions with different parameters using the graphs in Grafana.

7 Results & Discussion

This chapter presents the results of the experiments and discusses their relevance and limitations.

7.1 GMT Results

Table 7.1 shows the results of eight measurement runs with the GMT, four with the modulith and four with the microservices system, using different configurations. There are two parameters that differ between the individual measurement runs: Number of sequential executions (0, 1 or 100) and think time between executions (0 or 1 second). In these scenarios, there is only ever one user who makes sequential requests, hence there are no parallel accesses. Think time is used as a parameter to include measurements when the machine is idle.

Table 7.1: Results of eight measurement runs with the Green Metrics Tool.

	Number of Executions	Think Time [s]	Duration [s]	Machine Energy [J]	CPU Energy [J]	Memory Energy [J]	Network Energy [J]	SCI [mg CO_2e / order]
Modulith[1]	0	0	3.81	113.25	53.19	3.00	0.00	N/A
Microservices[2]	0	0	4.01	125.64	55.92	3.36	0.46	N/A
Modulith[3]	1	0	5.82	181.52	85.83	5.40	1.02	34.2
Microservices[4]	1	0	16.56	441.30	164.29	14.00	4.74	87.3
Modulith[5]	100	0	13.40	393.86	166.47	13.51	83.08	0.8
Microservices[6]	100	0	78.13	1815.27	572.10	67.15	330.28	3.8
Modulith[7]	100	1	113.50	1809.81	254.30	58.08	84.94	4.4
Microservices[8]	100	1	175.16	3457.65	808.74	115.46	345.57	7.6

[1] https://metrics.green-coding.io/stats.html?id=f1e0171c-a5f6-4f24-b5e4-558fe334993c
[2] https://metrics.green-coding.io/stats.html?id=e6c84f8f-971e-4401-97b1-3cd75e57c4a9
[3] https://metrics.green-coding.io/stats.html?id=25614e23-d474-4953-a08b-3808f8e46fe6
[4] https://metrics.green-coding.io/stats.html?id=59ed4330-d15b-465f-933c-9a7d966802f0
[5] https://metrics.green-coding.io/stats.html?id=7e40ee3b-733e-4b66-aaba-e1e32a412a28
[6] https://metrics.green-coding.io/stats.html?id=bf22a5c1-670b-4bd2-ba94-ad225cefe7c0
[7] https://metrics.green-coding.io/stats.html?id=c8aca13e-428a-4616-8677-93db8ebb0259
[8] https://metrics.green-coding.io/stats.html?id=a372a5ed-cb11-45bd-9b4c-8ad626f451bd

The measurement run in the microservices system with one execution (forth row in the table) took significantly longer than expected. This was due to the fact that JMeter took around ten seconds to start instead of the usual three seconds. A repeated measurement run[9] also resulted in the same behavior. This shows that JMeter behaves unevenly at the start, which can have a negative impact on the results. The reasons for this are unclear.

After the measurement runs, some small changes were made to the T2-Project, such as an update from Spring Boot 3.1 to 3.2, an optimization of the database query for reserving items in the shopping cart and a change in the configuration of the Eventuate CDC service from polling to Postgres-WAL. A new measurement run in the scenario with 100 executions without think time on the T2-Microservices system[10] resulted in a runtime of 79 seconds and a machine energy consumption of 1821 J, which shows that the changes had no significant impact on energy consumption.

Some load test scenarios were also tried out with the Green Metrics Tool, but with the same end result: the monolith performs significantly better. However, this is hardly surprising, as horizontal scaling is not possible in the setup with GMT and therefore the microservices cannot utilize their potential advantages in this respect.

7.2 Kepler Results

Unfortunately, no representative measurements could be carried out with Kepler within the time frame of this thesis.

However, the test measurements carried out so far indicate that a comparatively high load is required before the microservices system can make up for its disadvantages in terms of energy efficiency and potentially perform better than the monolithic system.

7.3 Discussion

The results of the measurements with the Green Metrics Tool show impressively that the monolithic system performs significantly better in the measured scenarios without scaling. This is mainly due to the fact that the microservices system takes longer to respond to the requests. For example, the scenario with 100 requests without think time only took 13 seconds for the monolith, while it took 78 seconds for the microservices system. The longer response times can be explained in particular by the overhead of network communication, as other studies comparing the performance of the two architectural styles have already shown (see Chapter 3 “Related Work”).

The tested microservices system also performs worse at idle, as it causes a greater overall overhead due to the larger number of services and runtime environments, more databases and the additional middleware required for the message-based communication needed for saga.

[9] https://metrics.green-coding.io/stats.html?id=e8f20671-32f3-44f8-9710-c665a42ae036
[10] https://metrics.green-coding.io/stats.html?id=bcfc1dd5-f93b-4cd0-9ceb-330c8b02ac3c

It is not possible to say how relevant the aspect of distributed transactions with saga is for the microservices system based on the measurements carried out. Saga operations are processed asynchronously and are therefore not directly responsible for the higher response times.

The existing studies on performance comparison have shown that the microservices system can perform better as soon as scaling is required. Corresponding measurements with scaling scenarios in terms of energy efficiency were planned with Kepler, but could not be carried out in the end as part of this thesis. A final conclusion can therefore not be drawn.

It is interesting to see that the cloud providers recommend fine-grained architectures in their recommendations, which is understandable given that these tend to better exploit the advantages of cloud computing. However, this also shows that the recommendations are not suitable for all application scenarios and should be treated with caution.

7.4 Threats to Validity

The classification by Wohlin et al. [WRH+12] is used to describe the threats to validity.

Construct validity The test environment with GMT differs in some respects from a productive environment with Kubernetes, for example. In particular, since no scaling behavior can be taken into account with GMT, the results can only be used to a limited extent to draw conclusions. For this reason, another test environment was created with Kepler and a Kubernetes cluster, which comes very close to a productive environment. However, compared to GMT, this test environment makes it much more difficult to measure the energy efficiency of usage scenarios.

Another problem with the test setup is the pure focus on energy consumption without taking costs into account. In reality, costs play an important role, so they should also be taken into account when considering energy efficiency.

Internal validity There are some aspects of the experiments that are out of control and can unintentionally influence the results. This applies in particular to the measurements with Kepler in the Kubernetes cluster. JMeter is run locally on a laptop, so access is via the internet. This means that network fluctuations can influence the results. The decision to run JMeter locally is in line with the philosophy of mapping user behavior as realistically as possible. The cloud resources used can also vary somewhat in their performance despite the guaranteed resources. Kepler is also not designed to capture the energy consumption of a single scenario with short runtimes. Instead, Kepler is much more suitable for recording the energy consumption of workloads over a longer period of time. The situation is different in the experiments with the Green Metrics Tool. Here, the Green Coding Solutions team ensures that the machines used are calibrated and that the measurement results do not deviate significantly from one another between runs. However, fluctuations are also conceivable here, so that multiple runs and statistical methods should be used to improve the results (see conclusion validity).

External validity The T2-Project is a very small application that represents a single scenario and can therefore hardly be compared with real applications in terms of its size and complexity. In particular, it is questionable how well the sole comparison of a scenario with saga (microservices) and a scenario without (monolith) is suitable as a representative comparison for microservices vs. modular monolith.

Conclusion validity No statistical methods were used to check the validity of the results. The sample size for the experiments is also very small. The results of this thesis should therefore be viewed with great caution.

8 Conclusion & Future Work

Finally, a summary of this thesis, a description of potentially interested parties, limitations of the study, important findings and suggestions for future research.

Summary

Microservices architectures are often used in cloud-based environments, as they promise to potentially make the best use of the advantages of cloud computing. Cloud providers also recommend migrating from monolithic systems to fine-grained architectures. However, measurements with regard to energy efficiency show that the microservices architecture style also has disadvantages that can potentially have a negative impact on energy efficiency, depending on the scenario. This is particularly the case in scenarios with a low load or with a constantly high load where no scaling is required. In such scenarios, monolithic systems have an advantage and can offer better performance and better energy efficiency. This thesis confirms this by using measurements carried out with the Green Metrics Tool.

Microservices have an advantage when the system needs to be scaled dynamically. The advantage increases if the services are scaled differently, for example due to compute-intensive operations. In these scenarios, a microservices system can be scaled in a fine-grained manner, whereas a monolith can only be scaled with significantly more overhead. Existing studies on performance measurements confirm this.

The microservices architecture style is also popular because it is associated with modularity and thus promises improved maintainability, among other things. However, a modular architecture can also be achieved with a monolithic system, as shown by the migration from a microservices system to a modular monolith as part of this thesis. The advantage of this is that modularity is achieved without the complexity of a distributed system.

There is no general answer to the question of which architecture should be chosen for an energy-efficient and modular system. Every application requires different compromises, so decisions have to be made individually. However, two rough rules can be derived from this thesis that can help with individual architectural decisions:

- Modularity can be realized with both a monolithic and a microservices architecture style, but with less complexity in the monolithic system.
- Scalability plays an important role in the architectural decision regarding performance and energy efficiency. Striving for the best possible scalability without having the need for it has a potentially negative impact on performance and energy efficiency. However, if dynamic and high loads are expected and good scalability is required, fine-grained architectures have an advantage.

Benefits

Software architects in particular can benefit from this thesis in order to be able to make adequate architectural decisions with the goals of modularity and energy efficiency. In addition, the measurement setups described can be helpful for others who want to carry out similar measurements.

Limitations

The goal of developing a software application that is more environmentally friendly, or at least more climate-friendly, requires more than just looking at energy efficiency. In particular, the aspect of embodied emissions must not be ignored when considering the carbon footprint, as this can have a significant impact depending on the application. Furthermore, there are numerous other sustainability aspects that should also be included in a holistic approach.

Too few measurements were taken with Kepler as part of this thesis, so that ultimately no representative results could be presented with regard to the aspect of scalability. The statements made on energy efficiency in scaling systems are based on the findings of existing studies on performance. It is necessary to carry out concrete measurements on energy efficiency in which scaling behavior plays a role.

The question arises as to how representative the reference application used, T2-Project, is, which consists of only one application scenario and uses the microservices variant saga to achieve transactional integrity. Accordingly, it makes sense to carry out further measurements with other application types.

Lessons Learned

The T2-Project was chosen as the reference application for this thesis, in particular because it was developed at the same institute. It would have been helpful for this thesis if there had already been similar studies, e.g. on performance, as is the case with other comparable microservices reference applications such as TeaStore or DeathStarBench. The applications that were used once in the performance studies (discussed in the Related Work section) would also have had the advantage of being able to compare the results. This was not the case with the T2-Project.

Ensuring transactional integrity in a distributed system is a challenge. The T2-Project explicitly aims to ensure this and uses the saga pattern and the transactional outbox pattern in the microservices variant for this purpose. In addition to PostgreSQL databases, MongoDB is also used, also as part of a saga operation. However, as the Eventuate Tram Framework does not support MongoDB, no transactional outbox pattern is used here. This should briefly illustrate that the topic is complex and it is difficult to avoid all the pitfalls. When migrating from the microservice system to the monolithic system, the MongoDB instance was kept, but this turned out to be a mistake. The retention of a PostgreSQL and a MongoDB database makes the execution of a local transaction impossible.

Knowledge and experience in the field of performance engineering and benchmarking are very helpful for evaluating the energy efficiency of software. Not having the relevant prior knowledge was a major disadvantage in this thesis. Based on this experience, it can be said that it is advisable to look into performance engineering and benchmarking and not just focus purely on theories for measuring energy consumption.

As far as the comparison of the measurement tools used is concerned, it can be concluded that the Green Metrics Tool is highly recommended if specific usage scenarios are to be measured and scaling behavior does not play a decisive role. Kepler, on the other hand, is well suited if a Kubernetes cluster is already in use and therefore little effort is required to measure the energy consumption of the workloads.

Future Work

The prepared measurements with Kepler should still be completed. This would greatly help to better evaluate the energy efficiency of the two variants in terms of scaling behavior. This would possibly also answer the question of when and under what specific conditions the microservices variant performs in a more energy-efficient manner.

This thesis has only examined the energy efficiency of two architectural styles using one example. Future research should look at other aspects in different ways. These include, among others:

- Comparison with other application scenarios
- Comparison with more serverless applications
- Comparison of different runtime environments
- Comparison of different frameworks
- Comparison of energy-awareness strategies
- Comparison with other sustainability aspects

Bibliography

[ABPL22] V. Andrikopoulos, R.-D. Boza, C. Perales, P. Lago. "Sustainability in Software Architecture: A Systematic Mapping Study". In: *2022 48th Euromicro Conference on Software Engineering and Advanced Applications (SEAA)*. 2022 48th Euromicro Conference on Software Engineering and Advanced Applications (SEAA). Aug. 2022, pp. 426–433. DOI: 10.1109/SEAA56994.2022.00073. URL: https://ieeexplore.ieee.org/document/10011515 (visited on 03/08/2024) (cit. on p. 2).

[ACC+23] M. Amaral, H. Chen, T. Chiba, R. Nakazawa, S. Choochotkaew, E. K. Lee, T. Eilam. "Kepler: A Framework to Calculate the Energy Consumption of Containerized Applications". In: *2023 IEEE 16th International Conference on Cloud Computing (CLOUD)*. 2023 IEEE 16th International Conference on Cloud Computing (CLOUD). July 2023, pp. 69–71. DOI: 10.1109/CLOUD60044.2023.00017. URL: https://ieeexplore.ieee.org/document/10254956 (visited on 02/01/2024) (cit. on p. 31).

[AM18] O. Al-Debagy, P. Martinek. "A Comparative Review of Microservices and Monolithic Architectures". In: *2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI)*. 2018 IEEE 18th International Symposium on Computational Intelligence and Informatics (CINTI). Nov. 2018, pp. 000149–000154. DOI: 10.1109/CINTI.2018.8928192. URL: https://ieeexplore.ieee.org/document/8928192 (visited on 03/22/2024) (cit. on p. 36).

[Ama23] Amazon Web Services. *AWS Well-Architected Framework - Sustainability Pillar*. Oct. 3, 2023. URL: https://docs.aws.amazon.com/wellarchitected/latest/sustainability-pillar/sustainability-pillar.html (visited on 01/24/2024) (cit. on pp. 7, 15).

[AMKF18] J. Aslan, K. Mayers, J. G. Koomey, C. France. "Electricity Intensity of Internet Data Transmission: Untangling the Estimates". In: *Journal of Industrial Ecology* 22.4 (2018), pp. 785–798. ISSN: 1530-9290. DOI: 10.1111/jiec.12630. URL: https://onlinelibrary.wiley.com/doi/abs/10.1111/jiec.12630 (visited on 04/04/2023) (cit. on p. 21).

[BAB+21] N. Bjørndal, L. Araújo, A. Bucchiarone, N. Dragoni, M. Mazzara, S. Dustdar. "Benchmarks and Performance Metrics for Assessing the Migration to Microservice-Based Architectures". In: *Journal of Object Technology* (Aug. 1, 2021). DOI: 10.5381/jot. URL: https://www.researchgate.net/publication/353337921_Benchmarks_and_performance_metrics_for_assessing_the_migration_to_microservice-based_architectures (cit. on pp. 36, 40).

[BCK21] L. Bass, P. Clements, R. Kazman. *Software Architecture in Practice*. Fourth edition. Addison-Wesley, 2021. ISBN: 978-0-13-688567-2 (cit. on pp. 13, 28).

[BHR19] L. A. Barroso, U. Hölzle, P. Ranganathan. *The Datacenter as a Computer: Designing Warehouse-Scale Machines*. Synthesis Lectures on Computer Architecture. Cham: Springer International Publishing, 2019. ISBN: 978-3-031-01761-2. DOI: 10.1007/978-3-031-01761-2. URL: https://link.springer.com/10.1007/978-3-031-01761-2 (visited on 07/12/2023) (cit. on pp. 19–21).

[BOP22] G. Blinowski, A. Ojdowska, A. Przybylek. "Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation". In: *IEEE Access* 10 (2022), pp. 20357–20374. ISSN: 2169-3536. DOI: 10.1109/ACCESS.2022.3152803. URL: https://ieeexplore.ieee.org/document/9717259/ (visited on 06/01/2023) (cit. on pp. 36, 40).

[Bri22] Y. Brikman. *How to Manage Terraform State*. Medium. Oct. 11, 2022. URL: https://blog.gruntwork.io/how-to-manage-terraform-state-28f5697e68fa (visited on 03/28/2024) (cit. on p. 60).

[Bro18] S. Brown. "Modular Monoliths". Conference Talk. GOTO 2018 (Berlin). Nov. 22, 2018. URL: https://youtu.be/5OjqD-ow8GE (visited on 03/16/2024) (cit. on pp. 10, 11).

[CB19] J. A. de Chalendar, S. M. Benson. "Why 100% Renewable Energy Is Not Enough". In: *Joule* 3.6 (June 19, 2019), pp. 1389–1393. ISSN: 2542-4785, 2542-4351. DOI: 10.1016/j.joule.2019.05.002. URL: https://www.cell.com/joule/abstract/S2542-4351(19)30214-4 (visited on 03/08/2024) (cit. on p. 2).

[CdB23] L. Cruz, P. de Bekker. *All You Need to Know about Energy Metrics in Software Engineering*. Luís Cruz. 2023. URL: http://luiscruz.github.io/2023/05/13/energy-units.html (visited on 03/08/2024) (cit. on pp. 17–19).

[Clo24] G. Cloud. *Go Green Software Guide*. Google Cloud, 2024. URL: https://inthecloud.withgoogle.com/go-green-engineering/guide.html (visited on 02/15/2024) (cit. on p. 16).

[CMG+20] C. Calero, J. Mancebo, F. García, M. Á. Moraga, J. A. G. Berná, J. L. Fernández-Alemán, A. Toval. "5Ws of Green and Sustainable Software". In: *Tsinghua Science and Technology* 25.3 (June 2020), pp. 401–414. ISSN: 1007-0214. DOI: 10.26599/TST.2019.9010006 (cit. on p. 2).

[CPM21] C. Calero, M. Polo, M. Á. Moraga. "Investigating the Impact on Execution Time and Energy Consumption of Developing with Spring". In: *Sustainable Computing: Informatics and Systems* 32 (Dec. 1, 2021), p. 100603. ISSN: 2210-5379. DOI: 10.1016/j.suscom.2021.100603. URL: https://www.sciencedirect.com/science/article/pii/S2210537921000913 (visited on 03/17/2023) (cit. on p. 3).

[Dee23] A. Deenadayalan. *AWS Prescriptive Guidance - Cloud Design Patterns, Architectures, and Implementations*. Nov. 16, 2023. URL: https://docs.aws.amazon.com/prescriptive-guidance/latest/cloud-design-patterns (visited on 01/25/2024) (cit. on p. 17).

[DLW22] K. Dürr, R. Lichtenthäler, G. Wirtz. "Saga Pattern Technologies: A Criteria-Based Evaluation". In: *Proceedings of the 12th International Conference on Cloud Computing and Services Science*. 12th International Conference on Cloud Computing and Services Science. SCITEPRESS - Science and Technology Publications, 2022, pp. 141–148. ISBN: 978-989-758-570-8. DOI: 10.5220/0010999400003200. URL: https://www.scitepress.org/DigitalLibrary/Link.aspx?doi=10.5220/0010999400003200 (visited on 01/25/2024) (cit. on p. 14).

[EL21] J. Ehneß, S. Luber. *Was ist ein Hyperscaler?* Storage-Insider. Feb. 9, 2021. URL: https://www.storage-insider.de/was-ist-ein-hyperscaler-a-986939/ (visited on 03/16/2024) (cit. on p. 6).

[Eur23] European Environment Agency. *Greenhouse Gas Emission Intensity of Electricity Generation*. Sept. 13, 2023. URL: https://www.eea.europa.eu/data-and-maps/daviz/co2-emission-intensity-14/#tab-chart_6 (visited on 03/11/2024) (cit. on p. 23).

[Eva15] E. Evans. *Domain-Driven Design Reference: Definitions and Patterns Summaries*. Domain Language. Indianapolis, IN: Dog Ear Publishing, 2015. 75 pp. ISBN: 978-1-4575-0119-7 (cit. on p. 10).

[FBW+21] C. Freitag, M. Berners-Lee, K. Widdicks, B. Knowles, G. S. Blair, A. Friday. "The Real Climate and Transformative Impact of ICT: A Critique of Estimates, Trends, and Regulations". In: *Patterns* 2.9 (Sept. 10, 2021), p. 100340. ISSN: 2666-3899. DOI: 10.1016/j.patter.2021.100340. URL: https://www.sciencedirect.com/science/article/pii/S2666389921001884 (visited on 05/16/2022) (cit. on p. 1).

[FGPR24] D. Faustino, N. Gonçalves, M. Portela, A. Rito Silva. "Stepwise Migration of a Monolith to a Microservice Architecture: Performance and Migration Effort Evaluation". In: *Performance Evaluation* 164 (May 1, 2024), p. 102411. ISSN: 0166-5316. DOI: 10.1016/j.peva.2024.102411. URL: https://www.sciencedirect.com/science/article/pii/S0166531624000166 (visited on 03/22/2024) (cit. on pp. 37, 41).

[FL23] M. Funke, P. Lago. "Carving Sustainability into Architecture Knowledge Practice". In: *Software Architecture*. Ed. by B. Tekinerdogan, C. Trubiani, C. Tibermacine, P. Scandurra, C. E. Cuesta. Vol. 14212. Lecture Notes in Computer Science. Cham: Springer Nature Switzerland, 2023, pp. 54–69. ISBN: 978-3-031-42592-9. DOI: 10.1007/978-3-031-42592-9_4. URL: https://link.springer.com/10.1007/978-3-031-42592-9_4 (cit. on p. 2).

[Fou22] G. S. Foundation. *Software Carbon Intensity (SCI) Specification*. GitHub. 2022. URL: https://github.com/Green-Software-Foundation/sci/blob/main/Software_Carbon_Intensity/Software_Carbon_Intensity_Specification.md (visited on 03/11/2024) (cit. on pp. 26, 27).

[Fou24] G. S. Foundation. *Green Software Foundation Manifesto*. 2024. URL: https://greensoftware.foundation/manifesto (visited on 03/18/2024) (cit. on p. 26).

[FPG+23] J. Ferreira Loff, D. Porto, J. Garcia, J. Mace, R. Rodrigues. "Antipode: Enforcing Cross-Service Causal Consistency in Distributed Applications". In: *Proceedings of the 29th Symposium on Operating Systems Principles*. SOSP '23. New York, NY, USA: Association for Computing Machinery, Oct. 23, 2023, pp. 298–313. DOI: 10.1145/3600006.3613176. URL: https://dl.acm.org/doi/10.1145/3600006.3613176 (visited on 01/16/2024) (cit. on p. 13).

[FY97] B. Foote, J. Yoder. "Big Ball of Mud". In: *Pattern languages of program design* 4 (1997), pp. 654–692. URL: http://www.laputan.org/mud/ (cit. on p. 10).

[GBC+24] A. Guldner, R. Bender, C. Calero, G. S. Fernando, M. Funke, J. Gröger, L. M. Hilty, J. Hörnschemeyer, G.-D. Hoffmann, D. Junger, T. Kennes, S. Kreten, P. Lago, F. Mai, I. Malavolta, J. Murach, K. Obergöker, B. Schmidt, A. Tarara, J. P. De Veaugh-Geiss, S. Weber, M. Westing, V. Wohlgemuth, S. Naumann. "Development and Evaluation of a Reference Measurement Model for Assessing the Resource

and Energy Efficiency of Software Products and Components—Green Software Measurement Model (GSMM)". In: *Future Generation Computer Systems* 155 (June 1, 2024), pp. 402–418. ISSN: 0167-739X. DOI: 10.1016/j.future.2024.01.033. URL: https://www.sciencedirect.com/science/article/pii/S0167739X24000384 (visited on 03/24/2024) (cit. on pp. 32, 64).

[GBS17] D. Gannon, R. Barga, N. Sundaresan. "Cloud-Native Applications". In: *IEEE Cloud Computing* 4.5 (Sept. 2017), pp. 16–21. ISSN: 2325-6095. DOI: 10.1109/MCC.2017.4250939. URL: http://ieeexplore.ieee.org/document/8125550/ (cit. on p. 7).

[GGP+23] S. Ghemawat, R. Grandl, S. Petrovic, M. Whittaker, P. Patel, I. Posva, A. Vahdat. "Towards Modern Development of Cloud Applications". In: *Proceedings of the 19th Workshop on Hot Topics in Operating Systems*. HOTOS '23. New York, NY, USA: Association for Computing Machinery, June 22, 2023, pp. 110–117. DOI: 10.1145/3593856.3595909. URL: https://dl.acm.org/doi/10.1145/3593856.3595909 (visited on 01/17/2024) (cit. on p. 12).

[Goo21] Google. *Reduce Your Google Cloud Carbon Footprint*. Google Cloud Architecture Center. Oct. 11, 2021. URL: https://cloud.google.com/architecture/reduce-carbon-footprint (visited on 01/24/2024) (cit. on pp. 15, 16).

[Goo22] Google. *Transactional Workflows in a Microservices Architecture on Google Cloud*. Google Cloud Architecture Center. Apr. 4, 2022. URL: https://cloud.google.com/architecture/transactional-workflows-microservices-architecture-google-cloud (visited on 01/24/2024) (cit. on p. 17).

[Goo23] Google. *Interservice Communication in a Microservices Setup*. Google Cloud Architecture Center. Sept. 20, 2023. URL: https://cloud.google.com/architecture/microservices-architecture-interservice-communication (visited on 01/22/2024) (cit. on p. 17).

[Goo24] Google. *Carbon Footprint Reporting Methodology*. Carbon Footprint. Jan. 26, 2024. URL: https://cloud.google.com/carbon-footprint/docs/methodology (visited on 01/26/2024) (cit. on p. 25).

[GZC+19] Y. Gan, Y. Zhang, D. Cheng, A. Shetty, P. Rathi, N. Katarki, A. Bruno, J. Hu, B. Ritchken, B. Jackson, K. Hu, M. Pancholi, Y. He, B. Clancy, C. Colen, F. Wen, C. Leung, S. Wang, L. Zaruvinsky, M. Espinosa, R. Lin, Z. Liu, J. Padilla, C. Delimitrou. "An Open-Source Benchmark Suite for Microservices and Their Hardware-Software Implications for Cloud & Edge Systems". In: *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*. ASPLOS '19. New York, NY, USA: Association for Computing Machinery, Apr. 4, 2019, pp. 3–18. ISBN: 978-1-4503-6240-5. DOI: 10.1145/3297858.3304013. URL: https://dl.acm.org/doi/10.1145/3297858.3304013 (visited on 05/26/2023) (cit. on p. 38).

[HA15] L. M. Hilty, B. Aebischer. "ICT for Sustainability: An Emerging Research Field". In: *ICT Innovations for Sustainability*. Ed. by L. M. Hilty, B. Aebischer. Advances in Intelligent Systems and Computing. Cham: Springer International Publishing, 2015, pp. 3–36. ISBN: 978-3-319-09228-7. DOI: 10.1007/978-3-319-09228-7_1 (cit. on p. 5).

[IEA23] IEA. *Data Centres and Data Transmission Networks*. IEA. July 11, 2023. URL: https://www.iea.org/energy-system/buildings/data-centres-and-data-transmission-networks (visited on 03/05/2024) (cit. on p. 1).

[IEA24] IEA. *Electricity 2024 - Analysis and Forecast to 2026*. IEA. 2024. URL: https://www.iea.org/reports/electricity-2024 (visited on 03/19/2024) (cit. on p. 1).

[Ins11] W. R. Institute. *Corporate Value Chain (Scope 3) Accounting and Reporting Standard*. World Resources Institute, 2011. ISBN: 978-1-56973-772-9. URL: https://ghgprotocol.org/corporate-value-chain-scope-3-standard (visited on 03/09/2024) (cit. on p. 22).

[JDNK12] T. Johann, M. Dick, S. Naumann, E. Kern. "How to Measure Energy-Efficiency of Software: Metrics and Measurement Results". In: *2012 First International Workshop on Green and Sustainable Software (GREENS)*. 2012 First International Workshop on Green and Sustainable Software (GREENS). June 2012, pp. 51–54. DOI: 10.1109/GREENS.2012.6224256. URL: http://ieeexplore.ieee.org/document/6224256/ (cit. on p. 28).

[JV23] R. Jacob, L. Vanbever. "The Internet of Tomorrow Must Sleep More and Grow Old". In: *ACM SIGEnergy Energy Informatics Review* 3.3 (2023), pp. 27–32. DOI: 10.1145/3630614.3630620. URL: https://dl.acm.org/doi/10.1145/3630614.3630620 (visited on 03/23/2024) (cit. on p. 22).

[JVB+17] E. Jagroep, J. M. Van Der Werf, S. Brinkkemper, L. Blom, R. Van Vliet. "Extending Software Architecture Views with an Energy Consumption Perspective". In: *Computing* 99.6 (June 2017), pp. 553–573. ISSN: 0010-485X, 1436-5057. DOI: 10.1007/s00607-016-0502-0. URL: https://doi.org/10.1007/s00607-016-0502-0 (visited on 05/03/2023) (cit. on p. 31).

[KHG+18] E. Kern, L. M. Hilty, A. Guldner, Y. V. Maksimov, A. Filler, J. Gröger, S. Naumann. "Sustainable Software Products – Towards Assessment Criteria for Resource and Energy Efficiency". In: *Future Generation Computer Systems* 86 (Sept. 1, 2018), pp. 199–210. ISSN: 0167-739X. DOI: 10.1016/j.future.2018.02.044. URL: https://www.sciencedirect.com/science/article/pii/S0167739X17314188 (visited on 01/30/2023) (cit. on pp. 6, 27).

[KHN+18] K. N. Khan, M. Hirki, T. Niemi, J. K. Nurminen, Z. Ou. "RAPL in Action: Experiences in Using RAPL for Power Measurements". In: *ACM Transactions on Modeling and Performance Evaluation of Computing Systems* 3.2 (June 30, 2018), pp. 1–26. ISSN: 2376-3639, 2376-3647. DOI: 10.1145/3177754. URL: https://dl.acm.org/doi/10.1145/3177754 (visited on 06/21/2023) (cit. on p. 28).

[KZL+10] A. Kansal, F. Zhao, J. Liu, N. Kothari, A. A. Bhattacharya. "Virtual Machine Power Metering and Provisioning". In: *Proceedings of the 1st ACM Symposium on Cloud Computing*. SoCC '10. New York, NY, USA: Association for Computing Machinery, June 10, 2010, pp. 39–50. ISBN: 978-1-4503-0036-0. DOI: 10.1145/1807128.1807136. URL: https://dl.acm.org/doi/10.1145/1807128.1807136 (visited on 06/26/2023) (cit. on p. 30).

[LCD+23] H. Lee, K. Calvin, D. Dasgupta, G. Krinner, A. Mukherji, P. Thorne, C. Trisos, J. Romero, P. Aldunce, K. Barrett, G. Blanco, W. W. L. Cheung, S. L. Connors, F. Denton, A. Diongue-Niang, D. Dodman, M. Garschagen, O. Geden, B. Hayward, C. Jones, F. Jotzo, T. Krug, R. Lasco, J.-Y. Lee, V. Masson-Delmotte, M. Meinshausen, K. Mintenbeck, A. Mokssit, F. E. L. Otto, M. Pathak, A. Pirani, E. Poloczanska, H.-O. Pörtner, A. Revi, D. C. Roberts, J. Roy, A. C. Ruane, J. Skea, P. R. Shukla, R. Slade, A. Slangen, Y. Sokona, A. A. Sörensson, M. Tignor, D. van Vuuren, Y.-M. Wei, H. Winkler, P. Zhai, Z. Zommers. *Synthesis Report of the IPCC Sixth Assessment Report (AR6): Summary for Policymakers*. Intergovernmental Panel on Climate Change, 2023. URL: https://www.ipcc.ch/report/ar6/syr/downloads/report/IPCC_AR6_SYR_SPM.pdf (cit. on p. 1).

[LF14] J. Lewis, M. Fowler. *Microservices – a Definition of This New Architectural Term*. 2014. URL: https://martinfowler.com/articles/microservices.html (visited on 06/01/2023) (cit. on p. 10).

[LKM+13] P. Lago, R. Kazman, N. Meyer, M. Morisio, H. A. Müller, F. Paulisch. "Exploring Initial Challenges for Green Software Engineering: Summary of the First GREENS Workshop, at ICSE 2012". In: *ACM SIGSOFT Software Engineering Notes* 38.1 (Jan. 23, 2013), pp. 31–33. ISSN: 0163-5948. DOI: 10.1145/2413038.2413062. URL: https://dl.acm.org/doi/10.1145/2413038.2413062 (visited on 07/19/2023) (cit. on p. 2).

[LS23] C. Lilienthal, H. Schwentner. *Domain-Driven Transformation: Monolithen Und Microservices Zukunftsfähig Machen*. 1. Auflage. Heidelberg: dpunkt.verlag, 2023. 296 pp. ISBN: 978-3-86490-884-2 (cit. on pp. 8–10).

[LZJ+21] S. Li, H. Zhang, Z. Jia, C. Zhong, C. Zhang, Z. Shan, J. Shen, M. A. Babar. "Understanding and Addressing Quality Attributes of Microservices Architecture: A Systematic Literature Review". In: *Information and Software Technology* 131 (Mar. 2021), p. 106449. ISSN: 09505849. DOI: 10.1016/j.infsof.2020.106449. URL: https://www.sciencedirect.com/science/article/pii/S0950584920301993 (visited on 01/18/2024) (cit. on p. 13).

[LZS+21] R. Laigner, Y. Zhou, M. A. V. Salles, Y. Liu, M. Kalinowski. "Data Management in Microservices: State of the Practice, Challenges, and Research Directions". In: *Proceedings of the VLDB Endowment* 14.13 (Sept. 2021), pp. 3348–3361. ISSN: 2150-8097. DOI: 10.14778/3484224.3484232. URL: https://dl.acm.org/doi/10.14778/3484224.3484232 (visited on 01/16/2024) (cit. on p. 13).

[MBZ+16] I. Manotas, C. Bird, R. Zhang, D. Shepherd, C. Jaspan, C. Sadowski, L. Pollock, J. Clause. "An Empirical Study of Practitioners' Perspectives on Green Software Engineering". In: *Proceedings of the 38th International Conference on Software Engineering*. ICSE '16. New York, NY, USA: Association for Computing Machinery, May 14, 2016, pp. 237–248. ISBN: 978-1-4503-3900-1. DOI: 10.1145/2884781.2884810. URL: https://dl.acm.org/doi/10.1145/2884781.2884810 (visited on 07/19/2023) (cit. on p. 2).

[MG11] P. M. Mell, T. Grance. *The NIST Definition of Cloud Computing*. NIST SP 800-145. Gaithersburg, MD: National Institute of Standards and Technology, 2011, NIST SP 800–145. DOI: 10.6028/NIST.SP.800-145. URL: https://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf (visited on 01/29/2024) (cit. on p. 6).

[Mic23a] Microsoft. *Azure Well-Architected Framework - Mission-critical Workload Documentation*. Microsoft, Dec. 16, 2023. URL: https://learn.microsoft.com/en-us/azure/well-architected/mission-critical/ (visited on 01/24/2024) (cit. on p. 17).

[Mic23b] Microsoft. *Azure Well-Architected Framework - Sustainable Workload Documentation*. Microsoft, Dec. 7, 2023. URL: https://learn.microsoft.com/en-us/azure/well-architected/sustainability/ (visited on 01/24/2024) (cit. on pp. 7, 15, 16).

[MSL+20] E. Masanet, A. Shehabi, N. Lei, S. Smith, J. Koomey. "Recalibrating Global Data Center Energy-Use Estimates". In: *Science* 367.6481 (Feb. 28, 2020), pp. 984–986. DOI: 10.1126/science.aba3758. URL: https://www.science.org/doi/10.1126/science.aba3758 (visited on 03/23/2023) (cit. on p. 1).

[NDKJ11] S. Naumann, M. Dick, E. Kern, T. Johann. "The GREENSOFT Model: A Reference Model for Green and Sustainable Software and Its Engineering". In: *Sustainable Computing: Informatics and Systems* 1.4 (Dec. 1, 2011), pp. 294–304. ISSN: 2210-5379. DOI: 10.1016/j.suscom.2011.06.004. URL: https://www.sciencedirect.com/science/article/pii/S2210537911000473 (visited on 02/21/2023) (cit. on p. 5).

[OJ23] S. Okrój, P. Jatkiewicz. "Differences in Performance, Scalability, and Cost of Using Microservice and Monolithic Architecture". In: *Proceedings of the 38th ACM/SIGAPP Symposium on Applied Computing*. SAC '23. New York, NY, USA: Association for Computing Machinery, June 7, 2023, pp. 1038–1041. ISBN: 978-1-4503-9517-5. DOI: 10.1145/3555776.3578725. URL: https://dl.acm.org/doi/10.1145/3555776.3578725 (visited on 03/23/2024) (cit. on p. 37).

[ORRP20] Z. Ournani, R. Rouvoy, P. Rust, J. Penhoat. "On Reducing the Energy Consumption of Software: From Hurdles to Requirements". In: *Proceedings of the 14th ACM / IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM)*. ESEM '20. New York, NY, USA: Association for Computing Machinery, Oct. 5, 2020, pp. 1–12. ISBN: 978-1-4503-7580-1. DOI: 10.1145/3382494.3410678. URL: https://dl.acm.org/doi/10.1145/3382494.3410678 (visited on 09/14/2023) (cit. on p. 2).

[PC17] G. Pinto, F. Castor. "Energy Efficiency: A New Concern for Application Software Developers". In: *Communications of the ACM* 60.12 (Nov. 27, 2017), pp. 68–75. ISSN: 0001-0782, 1557-7317. DOI: 10.1145/3154384. URL: https://dl.acm.org/doi/10.1145/3154384 (visited on 06/19/2023) (cit. on p. 2).

[PHAH16] C. Pang, A. Hindle, B. Adams, A. E. Hassan. "What Do Programmers Know about Software Energy Consumption?" In: *IEEE Software* 33.3 (May 2016), pp. 83–89. ISSN: 0740-7459, 1937-4194. DOI: 10.1109/MS.2015.83. URL: https://ieeexplore.ieee.org/document/7155416/ (cit. on p. 2).

[PRRT14] B. Penzenstadler, A. Raturi, D. Richardson, B. Tomlinson. “Safety, Security, Now Sustainability: The Nonfunctional Requirement for the 21st Century”. In: *IEEE Software* 31.3 (May 2014), pp. 40–47. ISSN: 1937-4194. DOI: 10.1109/MS.2014.22 (cit. on p. 28).

[RBKW22] N. Rteil, R. Bashroush, R. Kenny, A. Wynne. “Interact: IT Infrastructure Energy and Cost Analyzer Tool for Data Centers”. In: *Sustainable Computing: Informatics and Systems* 33 (Jan. 2022), p. 100618. ISSN: 22105379. DOI: 10.1016/j.suscom.2021.100618. URL: https://www.sciencedirect.com/science/article/pii/S2210537921001062 (visited on 07/19/2023) (cit. on p. 31).

[Ric17a] C. Richardson. *Microservices Pattern: Database per Service*. microservices.io. 2017. URL: http://microservices.io/patterns/data/database-per-service.html (visited on 03/05/2024) (cit. on p. 10).

[Ric17b] C. Richardson. *Microservices Pattern: Saga*. microservices.io. 2017. URL: https://microservices.io/patterns/data/saga.html (visited on 01/17/2024) (cit. on p. 14).

[Ric17c] C. Richardson. *Microservices Pattern: Transactional Outbox*. microservices.io. 2017. URL: http://microservices.io/patterns/data/transactional-outbox.html (visited on 01/25/2024) (cit. on p. 14).

[RLQ23] G. Roussilhe, A.-L. Ligozat, S. Quinton. “A Long Road Ahead: A Review of the State of Knowledge of the Environmental Effects of Digitization”. In: *Current Opinion in Environmental Sustainability* 62 (June 1, 2023), p. 101296. ISSN: 1877-3435. DOI: 10.1016/j.cosust.2023.101296. URL: https://www.sciencedirect.com/science/article/pii/S187734352300043X (visited on 01/31/2024) (cit. on p. 5).

[ŠCR19] M. Štefanko, O. Chaloupka, B. Rossi. “The Saga Pattern in a Reactive Microservices Environment”. In: *Proceedings of the 14th International Conference on Software Technologies*. 14th International Conference on Software Technologies. Prague, Czech Republic: SCITEPRESS - Science and Technology Publications, 2019, pp. 483–490. ISBN: 978-989-758-379-7. DOI: 10.5220/0007918704830490. URL: http://www.scitepress.org/DigitalLibrary/Link.aspx?doi=10.5220/0007918704830490 (visited on 01/25/2024) (cit. on p. 14).

[SL24] R. Su, X. Li. *Modular Monolith: Is This the Trend in Software Architecture?* Jan. 22, 2024. DOI: 10.48550/arXiv.2401.11867. arXiv: 2401.11867 [cs]. URL: http://arxiv.org/abs/2401.11867 (visited on 03/21/2024). preprint (cit. on p. 11).

[Sol22] G. C. Solutions. *Philosophy & Methodology*. Green Metrics Tool. June 15, 2022. URL: https://docs.green-coding.io/docs/prologue/philosophy-methodology/ (visited on 03/26/2024) (cit. on p. 66).

[SSB22] S. Speth, S. Sties, S. Becker. “A Saga Pattern Microservice Reference Architecture for an Elastic SLO Violation Analysis”. In: *2022 IEEE 19th International Conference on Software Architecture Companion (ICSA-C)*. 2022 IEEE 19th International Conference on Software Architecture Companion (ICSA-C). Mar. 2022, pp. 116–119. DOI: 10.1109/ICSA-C54293.2022.00029. URL: https://ieeexplore.ieee.org/document/9779811/ (cit. on p. 3).

[SSS+16] A. Shehabi, S. Smith, D. Sartor, R. Brown, M. Herrlin, J. Koomey, E. Masanet, N. Horner, I. Azevedo, W. Lintner. "United States Data Center Energy Usage Report". In: (June 22, 2016). URL: https://escholarship.org/uc/item/84p772fc (visited on 03/23/2023) (cit. on p. 21).

[STF+21] A. Stephens, C. Tremlett-Williams, L. Fitzpatrick, L. Acerini, M. Anderson, N. Crabbendam. *Carbon Impact of Video Streaming*. Carbon Trust, 2021, p. 102. URL: https://www.carbontrust.com/our-work-and-impact/guides-reports-and-tools/carbon-impact-of-video-streaming (visited on 04/17/2023) (cit. on p. 21).

[Syn23] Synergy Research Group. *AI Helps to Stabilize Quarterly Cloud Market Growth Rate; Microsoft Market Share Nudges Up Again*. Strategic Market Intelligence for Emerging IT & Cloud. Oct. 26, 2023. URL: https://www.srgresearch.com/articles/ai-helps-to-stabilize-quarterly-cloud-market-growth-rate-microsoft-market-share-nudges-up-again (visited on 01/24/2024) (cit. on p. 7).

[Tar23a] A. Tarara. *Measure Energy Consumption for Each Individual Container / Reduce Overhead of Request Generators · Green-Coding-Solutions/Green-Metrics-Tool · Discussion #562*. GitHub. 2023. URL: https://github.com/green-coding-solutions/green-metrics-tool/discussions/562 (visited on 03/27/2024) (cit. on p. 66).

[Tar23b] A. Tarara. *Warming up of Applications · Green-Coding-Solutions/Green-Metrics-Tool · Discussion #595*. GitHub. 2023. URL: https://github.com/green-coding-solutions/green-metrics-tool/discussions/595 (visited on 03/27/2024) (cit. on p. 76).

[Tar24] A. Tarara. *CO2 Formulas*. Green Coding Solutions. 2024. URL: https://www.green-coding.io/co2-formulas/ (visited on 03/12/2024) (cit. on pp. 21, 75).

[Tho22] ThoughtWorks. *Methodology | Cloud Carbon Footprint*. Cloud Carbon Footprint. 2022. URL: https://cloud-carbon-footprint.github.io/docs/methodology/ (visited on 03/12/2024) (cit. on pp. 25, 26).

[TI17] C. Trust, G. e-Sustainability Initiative. *ICT Sector Guidance Built on the GHG Protocol Product Life Cycle Accounting and Reporting Standard*. World Resources Institute, 2017. URL: https://ghgprotocol.org/guidance-built-ghg-protocol (visited on 03/09/2024) (cit. on pp. 22, 29, 30).

[UNO16] T. Ueda, T. Nakaike, M. Ohara. "Workload Characterization for Microservices". In: *2016 IEEE International Symposium on Workload Characterization (IISWC)*. 2016 IEEE International Symposium on Workload Characterization (IISWC). Sept. 2016, pp. 1–10. DOI: 10.1109/IISWC.2016.7581269. URL: http://ieeexplore.ieee.org/document/7581269/ (cit. on p. 35).

[Vel24] I. Velasco. *Carbon Aware Computing: Next Green Breakthrough or New Greenwashing?* HackerNoon. Jan. 16, 2024. URL: https://hackernoon.com/carbon-aware-computing-next-green-breakthrough-or-new-greenwashing (visited on 03/12/2024).

[VES+18] J. Von Kistowski, S. Eismann, N. Schmitt, A. Bauer, J. Grohmann, S. Kounev. "TeaStore: A Micro-Service Reference Application for Benchmarking, Modeling and Resource Management Research". In: *2018 IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS)*. 2018 IEEE 26th International Symposium on Modeling, Analysis, and

Simulation of Computer and Telecommunication Systems (MASCOTS). Milwaukee, WI: IEEE, Sept. 2018, pp. 223–236. ISBN: 978-1-5386-6886-3. DOI: 10.1109/MASCOTS.2018.00030. URL: https://ieeexplore.ieee.org/document/8526888/ (visited on 05/04/2023) (cit. on p. 3).

[VGO+17] M. Villamizar, O. Garcés, L. Ochoa, H. Castro, L. Salamanca, M. Verano, R. Casallas, S. Gil, C. Valencia, A. Zambrano, M. Lang. "Cost Comparison of Running Web Applications in the Cloud Using Monolithic, Microservice, and AWS Lambda Architectures". In: *Service Oriented Computing and Applications* 11.2 (June 1, 2017), pp. 233–247. ISSN: 1863-2394. DOI: 10.1007/s11761-017-0208-y. URL: https://doi.org/10.1007/s11761-017-0208-y (visited on 03/23/2024) (cit. on p. 36).

[Vit22] M. Vitali. "Towards Greener Applications: Enabling Sustainable-aware Cloud Native Applications Design". In: *Advanced Information Systems Engineering*. Ed. by X. Franch, G. Poels, F. Gailly, M. Snoeck. Vol. 13295. Lecture Notes in Computer Science. Cham: Springer International Publishing, 2022, pp. 93–108. ISBN: 978-3-031-07472-1. URL: https://link.springer.com/10.1007/978-3-031-07472-1_6.

[VLVH22] S. Vos, P. Lago, R. Verdecchia, I. Heitlager. "Architectural Tactics to Optimize Software for Energy Efficiency in the Public Cloud". In: *2022 International Conference on ICT for Sustainability (ICT4S)*. 2022 International Conference on ICT for Sustainability (ICT4S). June 2022, pp. 77–87. DOI: 10.1109/ICT4S55073.2022.00019. URL: https://ieeexplore.ieee.org/document/9830087/ (cit. on p. 2).

[VSB23] M. Vitali, P. Schmiedmayer, V. Bootz. "Enriching Cloud-native Applications with Sustainability Features". In: *2023 IEEE International Conference on Cloud Engineering (IC2E)*. 2023 IEEE International Conference on Cloud Engineering (IC2E). Sept. 25, 2023, pp. 21–31. DOI: 10.1109/IC2E59103.2023.00011. URL: https://ieeexplore.ieee.org/document/10305842 (visited on 12/20/2023).

[Wes19] K. Westeinde. *Deconstructing the Monolith: Designing Software That Maximizes Developer Productivity*. Shopify. Feb. 21, 2019. URL: https://shopify.engineering/deconstructing-monolith-designing-software-maximizes-developer-productivity (visited on 03/22/2024) (cit. on p. 12).

[WM23] T. Ward, B. Mohan. *AWS Prescriptive Guidance - Enabling Data Persistence in Microservices*. Amazon Web Services, Dec. 4, 2023. URL: https://docs.aws.amazon.com/prescriptive-guidance/latest/modernization-data-persistence/welcome.html (visited on 01/24/2024) (cit. on p. 16).

[WRH+12] C. Wohlin, P. Runeson, M. Höst, M. C. Ohlsson, B. Regnell, A. Wesslén. *Experimentation in Software Engineering*. Berlin, Heidelberg: Springer, 2012. ISBN: 978-3-642-29044-2. DOI: 10.1007/978-3-642-29044-2. URL: http://link.springer.com/10.1007/978-3-642-29044-2 (visited on 03/28/2024) (cit. on p. 83).

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature