FAKULTÄT FÜR !NFORMATIK
Faculty of Informatics

Predictive Analytics for Smart Homes using Serverless Edge Computing

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieurin

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

David Schneiderbauer, BSc
Matrikelnummer 01026700

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Prof. Mag.rer.soc.oec. Dr.rer.soc.oec. Schahram Dustdar
Mitwirkung: Univ.Ass. Dipl.-Ing. Thomas Rausch, BSc

Wien, 1. April 2019

David Schneiderbauer   Schahram Dustdar

Predictive Analytics for Smart Homes using Serverless Edge Computing

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieurin

in

Software Engineering & Internet Computing

by

David Schneiderbauer, BSc
Registration Number 01026700

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Prof. Mag.rer.soc.oec. Dr.rer.soc.oec. Schahram Dustdar
Assistance: Univ.Ass. Dipl.-Ing. Thomas Rausch, BSc

Vienna, 1st April, 2019

David Schneiderbauer	Schahram Dustdar

Erklärung zur Verfassung der Arbeit

David Schneiderbauer, BSc
Dorf 57, 4751 Dorf an der Pram

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

Wien, 1. April 2019

David Schneiderbauer

Acknowledgements

First of all, I would like to say thank you to my supervisors Univ.Prof. Dr.rer.soc.oec. Schahram Dustdar and Thomas Rausch for their thorough supervision and guidance in all respects towards the finalization of this thesis. Above all, I owe particular thanks to Thomas Rausch for his constant perseverance and extremely valuable input into the direction of the thesis. Moreover, I greatly value his motivation and interest for the thesis and want to show my highest gratitude for his dedication throughout the whole time.
I also wish to state my sincere gratitude to my family: my parents Margarete and Helmut; my sisters Magdalena and Teresa; and my brother Lukas. My parents for their continuous support and foremost, for the possibility to deploy my prototype at their home and endure the tests of the consumption-oriented scheduling approach for more than a year. Additionally they provided the financial needs to upgrade the existing heating system with the required appliances. Thanks to his physics degree, Lukas was always available for all physical and mathematical issues I faced and provided the fundamental knowledge for the mathematical formulas.
Personally, it means a lot to me to express my appreciation and thank my girlfriend Eveline for her exceptional patience and understanding during this time. Her motivational speeches encouraged and pushed me to finally finish this diploma thesis.

Kurzfassung

Smart Home Technologien werden immer populärer und damit einhergehend immer öfter in unser tägliches Leben integriert. Das Internet of Things (IoT) ermöglicht die Kommunikation zwischen einer Vielzahl an smarten Geräten im Smart Home um mittels Predictive Data Analytics den Wohnkomfort zu verbessern und Haussysteme zu automatisieren. Zum größten Teil ist das heutige IoT mit Cloud Computing realisiert, weil es praktisch unlimitierte Rechen- und Speicherkapazität bereitstellt.
Die zentrale Cloud unterliegt aber auch einigen Einschränkungen: relativ hohe Latenzzeiten, hohe Bandbreitennutzung, limitierte Datenübertragungsrate und benötigt eine dauerhafte Internetverbindung. Edge Computing bringt Cloud Ressourcen näher zum Anwender und umgeht somit diese Einschränkungen. Dabei kommen heterogene, ressourcen-begrenzte Edge Geräte zum Einsatz auf welchen Applikationen ausgeführt werden können. Zur Unterstützung dieser heterogenen Hardware wurde das Serverless Programming Modell als ressourcenschonendes Abstraktionsmodell vorgeschlagen. Basierend darauf wurden bereits erste kommerzielle Edge Computing Frameworks von Cloud Providern entwickelt. Diese Frameworks ermöglichen die Ausführung von Serverless Funktionen auf Edge Geräten, die mittels Cloud Service konfiguriert und bereitgestellt werden. Es gibt aber nur wenige systematische Untersuchungen des Serverless Edge Computing Paradigmas anhand von praktischen Anwendungsfällen.
Das Ziel dieser Diplomarbeit ist die Evaluierung des Serverless Programming Modells von kommerziellen Frameworks im Kontext von IoT und Smart Home Data Analytics Applikationen. Dazu definieren wir einen repräsentativen IoT Anwendungsfall: Optimierung einer Warmwasserheizung. Anhand dessen Anforderungen analysieren wir verfügbare Frameworks, im Speziellen AWS Greengrass und Microsoft Azure IoT Edge. Basierend auf unserer Serverless Architektur für Smart Home Predictive Analytics wird ein Prototyp und eine cloud-basierte Referenz für die Evaluierung entwickelt. Schließlich evaluieren wir die Serverless Edge Computing Paradigma Behauptungen und präsentieren die Resultate unserer Optimierung.
Diese zeigen dass die Boilertemperatur und die Heizdauer bei gleichbleibendem Lebenskomfort erheblich reduziert werden können. Weiters stellt sich das Serverless Programming Modell als ein vielversprechendes Edge Abstraktionsmodell dar. Dennoch fehlt wichtige Funktionalität in kommerziellen Frameworks um die Edge Computing Vision zur Gänze umsetzen zu können. Code Mobility existiert nur sehr eingeschränkt und es ist nicht möglich Multi-Tenant Anwendungen bereitzustellen.

Abstract

Smart Home technologies become more and more popular and are increasingly integrated into our daily living environments. Smart Homes leverage the Internet of Things (IoT) and combine many smart devices to improve living comfort and automate essential supporting systems with predictive data analytics. For the most part, today's IoT is realized with cloud computing because it offers virtually unlimited computing and storage capabilities.

Despite its benefits the central cloud also introduces several limitations: relatively high latency, high Internet bandwidth usage, limited data transfer rate and requires a continuous Internet connection. To mitigate these limitations the edge computing paradigm brings data processing and storage closer to the user utilizing heterogeneous, resource-constrained edge devices. This considerably reduces latency and bandwidth usage and fosters scalability and reliability. To overcome the heterogeneity of edge devices the serverless programming model has been proposed as a light-weight computation model for the edge. Based on this proposal production-grade edge computing frameworks have been developed by cloud providers. These frameworks provide a serverless function orchestrator deployed at the edge and a cloud service for remote configuration and deployment of applications. However, there have only been few real-world use case evaluations that systematically examine the serverless edge computing paradigm.

The goal of this thesis is to evaluate the serverless programming model for production-grade edge computing frameworks in the context of IoT and Smart Home data analytics applications. First we define a representative real-world IoT use case: optimization of a domestic water heating system. Subsequently we examine available frameworks, specifically AWS Greengrass and Microsoft Azure IoT Edge, and choose one based on the use case requirements. Then, we outline a general serverless architecture for predictive Smart Home analytics applications. Based on this architecture a prototype and in addition a cloud-based baseline is implemented for the evaluation. Eventually we evaluate claims on the serverless edge computing paradigm and present use case results.

Our use case evaluation shows that the boiler temperature and heating time have been significantly reduced while still satisfying inhabitants' living comfort. Our evaluation of serverless edge claims indicates that the serverless programming model is a promising edge computation model. However, it also shows a lack of essential features in production-grade frameworks to fully realize the idea of edge computing. Foremost is the limited support of code mobility and missing support of multi-tenant application deployment.

Contents

CHAPTER 1

Introduction

1.1 Motivation

Ever since the emergence of the Internet of Things (IoT) the number of devices connected to the internet is rapidly growing and expected to exceed 24 billion devices by the year 2020 [GBMP13]. Those so called smart devices are built to simplify life by enabling remote control of everyday items. Smart things led in further consequence to the term Smart Home, a conglomeration of multiple connected devices in residences to increase living comfort through applications including but not limited to water heating control and smart meters. A key enabler for the IoT is the cloud computing paradigm [LL15]. Due to provided infinite on-demand computing power and storage it is an ideal solution to handle the vast amount of generated data. Furthermore the centralized cloud is easier to manage, allows more efficient scaling and enforces physical security [WSJ15]. For that reasons prevailing applications are usually implemented with a centralized cloud. In the meantime applications and frameworks were formed around the paradigm. One of them is the Amazon Web Services (AWS) IoT [Amab] Framework: a platform abstracting away relations between devices and the cloud providing configurable routing of messages via message brokers. Moreover the service takes care of managing infrastructure and uses auto-scaling techniques to cope with the load of billions of devices and messages. However topics like data privacy, security and handling of long-term connection-loss haven't been tackled extensively in current frameworks. Also latency sensitive applications, that require real-time processing of data in the order of a few milliseconds, form a problem in the centralized cloud approach [VS17].

Above issues strive for decentralization to satisfy their strict requirements, which in further consequence has led to an increase in research effort resulting in the edge computing architecture. [Sat17] states that in this arising paradigm resources are placed at the edge of the internet in close proximity to the users. Additionally it has been said that the dispersed nodes provide storage and computing power and facilitate highly responsive cloud services, scalability, privacy-policy enforcement and masking of cloud outages. This enables real-time data analytics at the edge including local storage of data in order to minimize sending data to cloud or perhaps even overseas. The heterogeneity of edge computing forces applications to be encapsulated in virtual machines or containers to be able to run on any node of the network. Latest scientific work [NRS+17] and in example AWS Greengrass [Amaa] focus on the serverless programming architecture to overcome the diversity. A description of serverless computing is given in [Eiv17] and says that instead of dedicated servers the focus in serverless computing is on light-weight, event-based functions, which are executed by an orchestration layer on-demand. Further it states that serverless functions can be thought of as encapsulated microservices in contrast to monolithic applications, and thus are well-suited for heterogeneous environments.

1.2 Problem Statement

The idea to apply the serverless programming model to the edge computing paradigm appears to be promising. Serverless programming offers a high abstraction level to the underlying hardware which removes infrastructure coupling. Furthermore it promotes code mobility and a decrease in infrastructure management complexity. Thus, software developers should be able to concentrate solely on the business logic. [BCC+17] These properties have been successfully proven to hold true for the cloud computing environment [CIMS17], but have yet to be put to the test for the edge computing paradigm. Researchers in this field have proposed various serverless edge analytics architectures [NRS+17, dLGL+16, MBS+17]. There have also been some experiments carried out with scientific implementations, e.g., see *Calvin Constrained* [MBS+17]. In the meantime commercial providers have developed proprietary frameworks that leverage this new tendency. However, it is unknown if these frameworks fully incorporate the edge computing architecture and the serverless programming model. Furthermore it has to be shown that real-world edge computing applications can be implemented without compromises with respect to their requirements within the serverless programming model. Advantages and disadvantages have to be shown in real-world applications. For this reasons, serverless programming at the edge research field is missing practical experiments and lack scientific evaluation.

1.3 Aim of the Work

To fill this identified research gap, the aim of the work is to evaluate the serverless architecture model for edge computing frameworks in the context of IoT and Smart Home Data Analytics applications. Such applications typically consist of several sensors, which measure certain metrics of interest, and various actuators to interact with local systems targeted by the application. These sensors usually generate lots of data which need to be processed and stored to perform real-time analytics as well as predictive analytics on historic data to later control actuators. Current approaches for IoT applications send the data into the cloud, apply data analytics routines and return control commands to the local system. Since this architecture is restricted by, e.g., available data transfer rate and a continuous internet connection, the edge computing paradigm is an essential key to overcome these issues and build more independent and stable IoT solutions. Edge computing enables local preprocessing of data, minimizes data transfer by filtering non-important data, can mask connection outages and furthermore enhances data privacy by reducing data amount to a minimum and by storing critical information locally. For that, existing IoT sensor/actuator gateways are typically extended with processing power and storage to enable the local execution of code. The serverless paradigm has emerged as a solution for an event-based execution environment on constrained IoT devices providing sandboxed environments for each application and execution on configurable events triggered in the network.

To showcase the viability of the edge computing paradigm we introduce a real-world IoT data analytics use case. Then, we implement this use case based on a generic edge data analytics architecture using a state-of-the-art edge computing framework leveraging serverless programming. Subsequently this prototype serves as a basis for the evaluation. This use case is briefly described as follows: Owners of a house want to minimize their energy costs used for providing warm water. This can be achieved on the one hand by regulating the heating cycles such that warm water is only heated when needed, and on the other hand by providing only the necessary amount of warm water instead of heating the water to maximum boiler temperature. In order to implement both optimizations, the house residents' warm water usage behavior has to be known in advance. Therefore the prototype will apply machine learning on historic data to model and predict consumption behavior and utilize real time analytics to take necessary actions on time.

We evaluate serverless programming in edge computing frameworks with respect to multiple dimensions. The architecture is compared to a native cloud solution in terms of quantitative and qualitative factors. Specifically, we examine financial aspects like fixed and variable costs of both solutions for the service provider. In addition to the economical interests data privacy is of great value nowadays and thus plays a central role when comparing both paradigms. With respect to the described use case of water heating optimization we finally discuss the possible savings resulting from the prototype compared to nowadays typical systems.

1.4 Methodology

To reach the expected results defined in Section 1.3 the methodological approach consists of four major parts. At first we conduct a literature review to gain knowledge in the concepts of edge computing and serverless programming. Subsequently we apply this knowledge when designing a serverless predictive edge analytics system, that is finally implemented in a prototype of our IoT use case. On completion we extract conclusive metrics during execution on our test environment — a genuine single-family home. Lastly we evaluate the results and complete the thesis with a discussion chapter and future research opportunities. The individual tasks are presented in the following:

1. **Literature review**

   In the first part we perform a literature review on the following topics: edge computing, serverless programming and data analytics. For a better understanding of the motivation towards the edge computing paradigm we additionally examine the IoT, current IoT application design and the utilization of IoT in Smart Homes. We further identify key challenges in state-of-the-art IoT application architectures and resulting limitations. With regard to the literature review on data analytics we narrow the search to the more specific topic of data analytics in IoT/Smart Homes.

2. **Use case definition**

   The second part introduces a Smart Home IoT use case hat we base our evaluation on. This use case has general elements of state-of-the-art IoT applications — sensors, actuators, predictive analytics and real-time data analytics. For this reason the use case can be used in the evaluation to argue about the serverless programming model at the edge. We break the use case chapter down into an introductory explanation of the current situation in household water heating systems to provide a common basis for the rest of the thesis. We then highlight several problems of the present situation and subsequently propose a solution.

3. **Design**

   Based on an evaluation on state-of-the-art edge computing frameworks the third part addresses the design and development of the prototype. The evaluation compares IoT frameworks for the edge based on key characteristics like deployment model, support of programming languages and the underlying security model. The introduced use case will then be implemented with one of these frameworks. Technically the prototype is outlined as follows: Each relevant device of the water heating system will get its own sensor/actuator implementation, called a „ thing“ in terms of AWS IoT. These things communicate with a central controller that temporarily stores received data in a local database. The controller also filters and aggregates the data and finally sends essential data into the cloud. This collected information is used on the one hand for predictive analytics on historic data and on the other hand for real-time analytics. Both results in conjunction enable the prediction of the next occurrence of warm water usage, the specific amount needed at that time and moreover prevent non-essential heat up of water.

4. **Evaluation**

   In the fourth and last part of the thesis we evaluate the prototype. We compare several aspects of edge computing using serverless architecture and cloud computing. In the evaluation we focus on multiple quantitative as well as qualitative aspects. For that reason we add comprehensive logging in our prototype to gather significant metrics. These metrics include the amount of sent messages between the devices, used Internet bandwidth and detailed measurements of execution times. Using the gained insights we discuss if proposed claims of the serverless programming model used in conjunction with edge computing hold true. Additionally we qualitatively evaluate the paradigm and discuss inter-host migrations of edge applications and edge device hardware replacement. Eventually we examine differences in data storage placement and discuss data privacy issues.

1.5 Structure of the Thesis

After the introduction chapter, that states the motivation behind this thesis and the aim of the work, we cover the fundamental concepts of IoT, Edge Computing and Serverless Programming in Chapter 2 *Background*. These lay the foundation for the research work presented in *Related Work* (see Chapter 3). We examine interdisciplinary research papers in these topics, specifically research in edge analytics platforms that leverage the serverless programming model. After having outlined the thesis' contributing fields we define the use case in *Use Case: Elastic Heat* (see Chapter 4). This use case presents a generalizable IoT scenario that is used as a basis for the evaluation and encompasses typical elements of an IoT application. It includes sensors, that persistently monitor the physical surrounding and additionally requires a predictive data analytics component that instructs actuators to act upon an application's goal. In Chapter 5 *Design* we lay out the system's architecture that is intended to solve the stated problems from Chapter 4. Furthermore, we address the development of our prototype and the consumption-oriented scheduling approach. We also describe our machine learning model that is capable of forecasting hourly hot water consumption required for accurate scheduling. Chapter 6 covers the *Evaluation* of the thesis. Therein we describe the cloud reference implementation at the beginning. Subsequently we compare the serverless edge analytics system to the cloud reference implementation by previously defined characteristics. Then corresponding results are presented and evaluated. A *Discussion* in Chapter 7 closes our thesis and summarizes the essential results. In the end future research opportunities in the interdisciplinary field of serverless programming model in edge computing are discussed.

CHAPTER 2

Background

This chapter covers the basics of all related topics to this diploma thesis. We begin with the vision of the IoT and necessary technologies to realize this vision. After that we go over to a practical example thereof, the Smart Home, and the application of analytical and predictive computations. Subsequently the edge computing paradigm is introduced. It is an alternative approach to the existing cloud computing model, in which computing resources are located geographically and/or logically closer to the end-user. Finally we present serverless programming, a new cloud computing model inbetween Software-as-a-Service (SaaS) and Platform-as-a-Service (PaaS).

2.1 Internet of Things

The IoT is a vision of ubiquitous computing, in which computing power is found in any form and everywhere. The goal of ubiquitous computing was to reduce complexity, demand of attention and isolation from social interactivity while interacting with PC's by creating an environment in which computers where seamlessly embedded in „the complex social framework of daily activity“ [WGB99]. In [AIM10] Atzori et al. extract three orientations in this vision of IoT. A things-oriented, an internet-oriented and a semantic-oriented perspective.

1. In the *things-oriented* perspective researchers primarily focus on the integration of everyday things into a common framework. Therefore they developed standards to improve objects visibility, enabling traceability and the exchange of objects state like its current location.

2. The *internet-oriented* perspective is concerned with the interconnectivity of things. They promote the „Internet Protocol as the network technology for connecting Smart Objects“ [AIM10] and therefore developed different lightweight variations of the current IP to drive the IoT.

3. The *semantic-oriented* perspective deals with issues arising from the vast amount of objects participating in the IoT. These include questions on how to uniquely address these things, how to store the massive amount of information and furthermore how to process or create meaning from this information.

Considering these three visions the „IoT can be considered as a global network infrastructure composed of numerous connected devices that rely on sensory, communication, networking, and information processing technologies“ [TW10]. It enriches the traditional internet, in which information was solely requested by humans and provided by web services to a network in which global information is also autonomously provided and consumed by a variety of appliances and electronic devices [WSJ15]. This shift in the internet is additionally pushed by some existing technologies like Wireless Sensor Network (WSN) and Radio-Frequency IDentification (RFID). In addition new low power communication technologies allow power-constrained devices to be incorporated into the IoT. This includes Bluetooth Low Energy (BLE), 6LoWPAN, ZigBee, RFID and Near Field Communication (NFC).

With all those connected devices, ranging from appliances in industry to private homes, the vision of ubiquitous and pervasive computing is close to realization. However there are still many open challenges that need to be solved. The massive amount of data that is generated by all those devices has to be efficiently stored and processed. Because IoT devices usually have limited computational power, storage and energy [AWW18] current implementations move control, storage and computational intensive tasks to the cloud [CZ16, DGC^{+}16]. The reason for that is cloud computing offers on-demand, virtual unlimited processing power and unlimited storage capacity, thus is a perfect fit for the ever-growing demand of IoT. This conjunction of IoT and cloud computing is termed Cloud of Things [AKAH14]. Furthermore Want et al. argue in [WSJ15] that centralized cloud services are easier to manage and maintain and also have the advantage of scale, automatic backup of data and enforced physical security of closed off data centers.

2.2 Smart Home

Besides applying the concept of IoT in the industry to support and reduce costs in manufacturing, its field of application also includes the domestic sector. One of the most prevalent use case is home automation, also referred to as Smart Home. A first definition of Smart Home or in general of Smart Environments was coined by Mark Weiser, a forefather of ubiquitous computing in 1999. He defines a Smart Environment as „a physical world that is richly and invisibly interwoven with sensors, actuators, displays, and computational elements, embedded seamlessly in the everyday objects of our lives, and connected through a continuous network“ [WGB99]. Back then the purpose of a Smart Environment was to improve the way we live and interact with computers, therefore the term „smart“ only referred to the integration of computing power into homes and socioenvironment of residents.

Today in addition to this socio-technical view Smart Homes are also dedicated to simplify life of its inhabitants, provide comfort and safety and reduce energy demand [RMD+06]. Aldrich presents a more up to date definition of a Smart Home in [Ald03]. He states that it is „a residence equipped with computing and information technology which anticipates and responds to the needs of the occupants, working to promote their comfort, convenience, security and entertainment through the management of technology within the home and connections to the world beyond“. Until now most of the existing consumer Smart Home solutions try to solve these tasks by offering a single intuitive interface for residents to change every aspect in homes according to their preferences [WRS+17]. Moreover the automation of home appliances happens only to a limited extent and is achieved by preconfigured scenarios and the respective actions the system has to execute. Typical scenarios are the management of space heating based on individual preferences and weather, management of illumination depending on the time of the day and brightness or saving of energy by automatically switching off unused devices.

2.2.1 Data Analytics & Machine Learning

In the recent years researchers focused on the realization of a truly automated home and continue to make progress utilizing artificial intelligence techniques. Ricquebourg et al. [RMD+06] claim Smart Home technology is not limited to turning devices on and off, but rather monitors the internal environment and activities that are being performed. For that purpose algorithms and statistical analysis methods from Big Data Analytics, Machine Learning and Artificial Intelligence are used to build up knowledge out of measured physical information. Applied in Smart Home these techniques allow the prediction of future actions conducted by inhabitants based on historical data and the forecast of residents' locations [WRS+17]. This data can be further used to automatically control home appliances based on observed habits and expected whereabouts of inhabitants rather than relying on a limited amount of preconfigured scenarios. Thus the notion of „smart“ transitioned in the last years from being equipped with processing power to being context aware and having the ability to fine-tune itself from observed behavior.

In addition to the algorithms that are used to predict future activities [AN06, WRS+17], Smart Home applications also perform time series forecasting to anticipate future values in temporal data [BBB+13]. Temporal data or time series refers to historical data measured at equal time intervals. Examples thereof are measurements of outdoor temperature measured every hour or hourly water consumption. For that purpose various machine learning models have been developed [BBB+13, AAGES10, Die02]. One of these models will be used in the prototype for this thesis to build the forecasting of warm water usage. The reliability of such forecasting models mainly depend on the available predictors and information granularity. Regularities in Smart Homes are „not only based on time of day and day of week but rather are based on a large number of factors“ [Moz05]. For that reason it is usually not sufficient to only sense the data you want to predict, but also gather information the predicted value depends on. Furthermore the granularity of temporal data determines what the system can achieve and how well it can perform.

Augusto et al. [AN06] state that in order to infer a trend it may be needed to view the data in bigger interval of several minutes or days.

2.3 Edge Computing

As already pointed out in Section 2.1, IoT devices are typically resource constrained devices with limited amount of computational power, battery, storage and bandwidth [AWW18, YLL15]. Furthermore Atlam et al. [AWW18] state that the IoT suffers from performance, security, privacy and reliability issues. The integration of cloud computing into IoT solves many of these issues [AAA$^+$17] but also implicates new challenges. Due to the combination of cloud and IoT issues like high round-trip time or latency, network bandwidth constraints, intermittent connectivity and new security implications have to be payed attention to [AWW18, CZ16, DB16, VS17]. Besides these technical challenges the cloud also implies problems from a user-centric perspective discussed in [GME$^+$15]. Moving all personal and social data generated by IoT devices to centralized services implies a loss of privacy. Moreover, having application logic reside in the cloud takes away control over the system from users and delegates control to the centralized service [GME$^+$15].

Subsequently, many researchers devoted themselves to develop a computing model that can solve these issues. As a result, various computing paradigms have been proposed, among them edge computing [SCZ$^+$16], fog computing [BMZA12] and cloudlet computing [Sat17]. However, literature presents many different conflicting definitions. Some research works clearly distinguish between edge and fog computing and argue that fog computing also leverages computation resources inbetween the edge and cloud in addition to the limited edge resources [VS17, DB16]. By contrast, others use *edge computing* as the umbrella term and call fog and cloudlet computing different implementations thereof [DD17, APZ18]. Additionally, Shi et al. [SCZ$^+$16] interpret the edge and fog computing paradigms equally and state that both terms can be used interchangeably. Likewise, Yi et al. [YHQL15] state that all of them describe eminently similar computing models that lack a common definition that abstracts all. Because most papers in this scientific field describe profoundly the same computation model no matter whether the authors discuss edge or fog computing, to the best of our knowledge, we adopt Shi et al.'s interpretation and use both terms interchangeably throughout this thesis. Consequently, we use the following definition of edge/fog computing by Bonomi et al. [BMZA12]: *„Fog computing is a highly virtualized platform that provides compute, storage, and networking services between end devices and traditional cloud computing data centers, typically, but not exclusively located at the edge of the network“*

The fundamental aspect of the edge computing paradigm is that computing should happen close to the data sources. [SCZ$^+$16] Nevertheless it is not a substitute of the cloud, but rather thought of as an extension of the cloud to the edge of the network [AWW18, BMNZ14, YMSG$^+$14]. The main characteristics of the fog computing paradigm are mobility support, location awareness, widely geographically distributed heterogeneous nodes and low latency [BMNZ14]. Therefore an edge computing platform should implement multi-tenant distributed deployment of applications across the path between the edge and the centralized cloud. Moreover the platform should handle „policy-based orchestration and provisioning for scalable and automatic resource management“ [BMNZ14]. This imposes the usage of virtualization techniques that isolate applications and enable inter-host migrations [BMNZ14]. Yanuzzi et al. claim that lightweight containers like LinuX Containers (LXC) [Lxc], Docker [Doc] and CRIU [Cri] are promising approaches especially in the context of IoT [YMSG+14]. In contrast, others propose the serverless architecture as an excellent model for infrastructure abstraction [NRS+17, dLGL+16, MBS+17]. Above all it also requires appropriate network virtualization like Software Defined Networking to support multi-tenant application deployments.

According to the stated definition, data storage, caching as well as data processing and analytics is shifted to the edge rather than being performed in a centralized cloud [BMZA12, SCZ+16, YMSG+14, DB16]. Moreover because it extends and doesn't substitute the cloud, components of edge computing applications run both in the cloud as well as in the edge devices, inbetween data sources and cloud data centers [DGC+16]. Alternatively to the dedicated providers' fog nodes, the fog layer also consist of computing resources from end users that want to share spare computation and storage resources of their private clouds, smart gateways or routers in return for compensation [YLL15, VRM14]. Figure 2.1 depicts the edge computing paradigm and the interplay between the edge and the cloud. Bonomi et al. describe the fog architecture as hierarchically structured from the edge to the cloud nodes in terms of information granularity and response time [BMNZ14]. The lowest level — the data producers / data consumers — perform real-time analytics on locally available data. Each further tier increases the scope of consumed data, e.g., data of a small city or later a geographical region, up to the global view at the cloud level. Moreover with each level latency between end devices increases as well.

The resultant benefits of edge computing are summarized in [DGC+16, CZ16, DD17]. Moving computational resources closer to the edge provides location-sensitive information of connected edge devices and thus enables local data processing based on geographical location [SCZ+16, AWW18]. Additionally it lowers latency of connections between end devices and edge computers — nodes that provide computational resources close to the edge [RAD] — and facilitates real-time response for time-sensitive applications. While the edge of the network changes due to the IoT from data-consumer only — retrieving information from servers — to also produce a compelling amount of sensory data [SCZ+16], Cisco predicts an increase in connected devices to around 50 billion devices by the year 2020 [Eva11]. Pushing the tremendous amount of data generated by all these devices to the cloud requires considerable network bandwidth. Edge computing enables „hierarchical data processing along the Cloud-to-Things continuum“ [AWW18] hence is able to filter and analyse generated data close to the edge before ever reaching the cloud. This hierarchical data processing drastically decreases traffic sent to the centralized cloud. As a consequence the fog provides scalability via edge analytics, since it also distributes

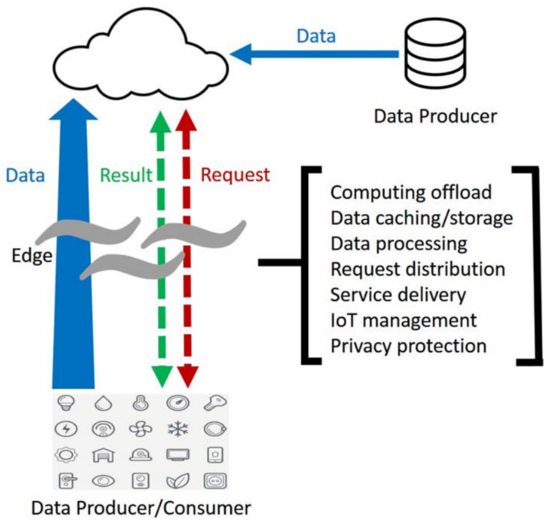


Figure 2.1: Edge computing paradigm [SCZ+16]

computational tasks between fog nodes and minimizes the cloud's workload. Additionally an edge computer is able to manage sporadic cloud service outages due to network failure, cloud failure or directed DOS attacks by operating autonomously thus masking cloud service outages. Moreover the fog node is also capable of executing resource-intensive tasks offloaded from resource-constrained devices on intermittent network connectivity.

2.4 Serverless Programming

As already stated edge computing requires a virtualization technique to isolate applications, enable multi-tenancy and live migration between nodes. In addition to the proposed options by Yanuzzi et al. [YMSG+14], LXC [Lxc], Docker [Doc] and CRIU [Cri], the work in [NRS+17] introduces the serverless programming model being a superior alternative. The authors argue that the serverless model has significant advantages in the context of cloud and edge computing because both try to reduce management and development effort in large-scale heterogeneous distributed networks.

The serverless computation is the latest derivative in the continuous development of server virtualization techniques and deployment of cloud applications [CIMS17]. Each progression increased the level of abstraction, from deployment on bare-metal machines to virtual machines and later to lightweight containers like LCX or Docker [HSH+16]. The reason for that is the difficulty in server configuration and management, furthermore efficient scaling is challenging because of considerable startup time [HSH+16]. The different types and evolution of deployment types are shown in Figure 2.2. The rightmost is the serverless programming model, also referred to as Lambda model because of AWS Lambda [Amac]. Compared to the other types serverless computation enables sharing of the runtime across applications in addition to the operating system for containerized applications and hardware in a virtualized environment.

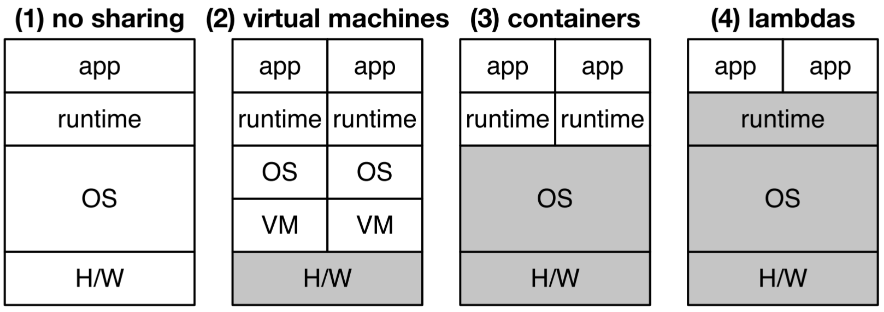


Figure 2.2: Evolution of sharing [HSH$^+$16]. *Gray layers are shared.*

In order to allow applications to use a shared runtime, they are decomposed into slim functions that are triggered by events. For that reason this deployment model is also called Function-as-a-Service (FaaS) following Infrastructure-as-a-Service (IaaS), PaaS and SaaS cloud offerings [CIMS17]. Therefore „instead of thinking of applications as collections of servers, developers instead define applications with a set of functions with access to a common data store“ [HSH$^+$16, BCF$^+$17]. These functions are stateless and thus require a shared data store for persistent information across multiple executions. During deployment of functions developers specify the desired runtime environment in which the code gets executed. Typically cloud providers only offer a limited amount of different environments like specific versions and programming languages. For that reason developers are forced to implement the business logic using one of the provided environments. Events trigger the execution of functions and mostly occur within the cloud provider's services [MB17]. This enables developers to build applications spanning over multiple cloud services.

Figure 2.3 depicts an example of a simple serverless application taken from [CIMS17]. The illustration shows a function whose single responsibility is to create thumbnails from uploaded images. In this scenario the image and thumbnail databases are object storages. On image upload the image database publishes a „new image“-event that triggers the user-defined function. Because functions are stateless they have to write the result back to a common data store, in this example the storage for the generated thumbnails. To deploy such „serverless application“ platforms must provide a way to manage these functions, as well as the association with events that trigger them.

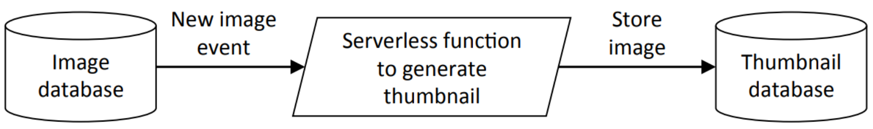


Figure 2.3: Example: Image processing using serverless programming [CIMS17]

This paradigm yields many benefits for developers as well as for cloud providers. Because functions in Serverless programming are stateless, application components can be scaled individually [MB17]. Moreover it enables scaling to zero, which consequently reduces costs for cloud applications because functions are billed by execution time and memory usage [Eiv17, CIMS17]. Another benefit of statelessness is the capability of cloud providers to transparently optimize and update the underlying system without service interference [BCC+17]. FaaS abstract away most operational concerns, thus freeing developers from system maintenance, considerations on scalability, elasticity and latency. Consequently developers can focus on business logic implementation rather than thinking about necessary cloud resources or virtual machine/container provisioning [BCC+17]. From a cloud provider's perspective being responsible for the underlying system allows efficient optimization and management of cloud resources, thus also reduces operational costs. Furthermore because user-defined functions run in predefined runtime environments they control the entire development stack and thus further improve manageability [CIMS17]. However FaaS providers may only support a limited amount of runtime environments, hence restricting developers in their choice of programming languages [BCC+17].

All these benefits are likely to hold true also for the serverless edge paradigm, hence it is thought of a good computing model for edge analytics. Especially beneficial is the lightweight nature of serverless functions, because it minimizes the overhead of virtualization compared to virtual machines and thus enables the execution on resource constrained IoT devices. Additionally most IoT applications are event-driven and perform actions on data arrival, which is a key aspect of the Serverless computation model [CIMS17].

CHAPTER 3

Related Work

This chapter discusses related research work surrounding the thesis' scientific fields. For that reason we outline the scientific fields to which this thesis contributes at the beginning of this chapter. Subsequently we introduce related work to give a fundamental overview of recent scientific work on these topics. Conclusively the divergence between this thesis and the presented work is discussed.

We have previously described the thesis' intentions in Section 1.3. The central designated aim of this work is a comparison between the nowadays commonly implemented cloud computing paradigm within the IoT context and modern edge computing architectures. Moreover we want to focus on the serverless programming model leveraged by recent edge frameworks. The use case introduced later in Chapter 4 by which we evaluate the paradigms is a general real-world IoT smart home scenario that requires data analytics on historic information as well as on real-time multi-sensory input. Despite involvement of machine learning algorithms in this particular use case we will not cover a comparison of available algorithms and an evaluation of the efficiency thereof. This summary of the thesis' aim yields several scientific fields to which this work contributes:

1. Edge computing
2. Serverless programming
3. Edge analytics
4. IoT / Smart Home

3.1 Serverless Edge Data Analytics Platforms

All of these scientific areas are reasonably novel, therefore receive great research effort. Numerous work about the serverless programming model within edge computing data analytics applications have been published in the recent years. Among them is [NRS+17] by Nastic et al.. The authors propose a „unified cloud and edge data analytics platform“ that implements the serverless programming model at the edge. Key aspect of the platform is a serverless stream model in which user-defined functions represent transformation functions along the data-stream. „Contracts“ enable the composition of these user-defined analytic functions to complex stream processing applications. For that reason they implement a 3-tier architecture consisting of a function wrapper layer, an orchestration layer and a runtime mechanisms layer. The former layer takes user-defined functions, adds a thin API and a state management component and then wraps each into a Linux container. This enables to individually scale these functions as needed. The orchestration layer interprets the contracts that specify deployment, placement and other runtime mechanisms. This layer decides how to deploy and execute the described function topologies using the mechanisms by the runtime mechanisms layer. The serverless stream model grants several benefits. First of all the serverless programming model provides a high abstraction of the execution environment so that user-defined functions can be deployed on any node from the edge to the cloud without effort. Consequently this uniform development model simplifies automatic orchestration of analytics functions in the heterogeneous edge infrastructure.

A similar approach was implemented by Lara et al. [dLGL+16]. They developed a research platform *EdgeScale* to explore the edge computing paradigm. Therefore they also adopted the serverless computation model. However in contrast to the work of Nastic et al. [NRS+17], the authors of EdgeScale did not pursue the idea of a serverless stream model. Instead they argue that the serverless programming paradigm supports „code and data mobility by enforcing a clear separation between computation and state“ and is thus an ideal computing model for the edge. Furthermore they state that stateless handlers typically are small and so easily migrated and executed on any node running EdgeScale. Analogous to the above data-stream analytics platform it also supports automatic deployment of applications and migrations between nodes with the overall aim of optimizing access latency and bandwidth consumption. However in addition EdgeScale further supports hardware acceleration services for, e.g., faster video and image analytics.

Similar to the two previous research works Mehta et al. [MBS+17] leverage the serverless programming model to ease „distributed application development, deployment and management on geo-distributed resources spanning from small constrained devices to servers in cloud Data Centers (DCs)“. Hence the authors developed *Calvin*, a framework for distributed IoT applications to overcome this research gap. The developers thereof implemented a variation of the serverless computation model for the IoT edge which they named *Actor-as-a-Service (AaaS)*. Contrary to serverless functions the AaaS features actors that represent individual application components like sensors, actuators or processing logic. However both functions and actors are stateless and self-contained. Actors can be composed to analytic applications by connecting them in a dataflow manner [EJ12]. For that reason another difference is that actors are triggered by tokens on their input ports or events from the hardware. Each participating node is equipped with the Calvin runtime, the execution environment, that provides platform abstraction and data transport facility. The Calvin framework implements automatic actor placement decision-making on the distributed execution environments based on actor requirements, e.g., a particular attached sensor and runtime capabilities. Furthermore it also performs auto-scaling of actors, which includes replication and migration thereof to different runtimes.

3.2 Edge Data Stream Analytics

In addition to the research work on edge data analytics platforms that aim to extend the serverless computing model to the IoT edge, researchers also explore more general IoT data stream analytics approaches with edge computing. One recent work in this field is [XHF$^+$17]. According to the researchers of this paper application development in the context of IoT requires *Edge Analytics-as-a-Service (EAaaS)*. Their conclusion is based on the prevailing high development effort for edge analytic logic along with immense expenses for manual management and monitoring of bespoke edge applications / infrastructure. Even simple analytic tasks as data stream aggregation require considerable expenditure. As a consequence they developed a platform that supports easy deployment of data analytic functions in a programming-free way. Their implementation is split up into cloud and gateway side components. A per-organization MQTT broker for message transfer, a real-time analytic service that allows management of rule-based analytic models and on top a RESTful API form the cloud side of the EAaaS platform. The counterpart is the Edge Analytic Agent consisting of a gateway controller, the core rule-based analytic engine and device adapters. Key aspect of the platform is the unified rule-based analytic model that provides a common formalization for real-time analytic logic. It facilitates easy utilization also for users that are not yet familiar with programming to realize analytic business logic. Moreover the model acts as a common analytic protocol that works on both cloud and edge nodes. Xu et al. extracted four common stages within analytic logics that are applied sequentially and specified in the analytic model: the data source and the required fields of interest for the particular analysis; various data transformations like mathematical functions and time-window based aggregations; rule conditions (data filtering); and subsequently actions that get triggered by the rule-hit. Actions can be user-defined local actions to third-party services or forwarding of events to the cloud service.

A different approach to edge analytics is taken by Renart et al. [RDMP17], however both [XHF$^+$17] and [RDMP17] share the idea of an abstraction of the analytics logic. In their article they propose a data-driven stream processing framework that allows users to define stream-processing workflows using conditions on the content of the streaming data. Basically the introduced framework consists of a peer-to-peer network connecting distributed heterogeneous resources — sensors, actuators and computational resources. Each computational resource runs a stream-processing engine responsible for applying user-defined workflows on arriving data. On top of the peer-to-peer network is an Associative Rendezvous Messaging Substrate (ARMS) for content-based decoupled interactions based on [JSMP04, JSP06] to discover available resources, data sources and services. Applications / Users interact with the network using a programming abstraction that defines „content-based interactions“ evaluated at runtime. In a first step the application requests available rendezvous points and selects one based on its requirements, e.g., low latency. Afterwards it can create/upload complex analytic workflows, also referred to as topologies that are executed on sent data. Every message in the network consists of a profile, a reactive behavior and optionally additional data, location and a topology. Profiles are either „interest profiles“ or „data profiles“ that are matched against each other. If a match occurred respective actions are executed. On an incoming message the rendezvous point performs service discovery to identify relevant services, data sources and computational resources taking user information — location, preferences, etc. — into account. This gathered information is then used by the routing component to „transparently decide where to perform the required computation“. Renart et al. argue that this approach enables location-aware computation of analytic workflows at the edge and furthermore is able to allocate streaming computations considering client location and data sources.

All these research articles tried to reduce the complexity of development and deployment of edge applications and also management and provision of distributed heterogeneous nodes along the path between the edge and the cloud. They mainly differ in the level of abstraction provided for developers / end users. [XHF+17] primarily provides a programming-free experience to end users, whereas [NRS+17, dLGL+16, MBS+17] provide a stripped down programming abstraction in the form of serverless functions or actors. A tremendous difference to these articles represents the approach of [RDMP17] that utilizes a peer-to-peer network consisting of edge and cloud nodes, and a well-defined message protocol to enable data-driven edge analytics. In contrast to these research works other scientific efforts in this field try to implement IoT scenarios in an edge computing fashion and evaluate improvements in the particular use case.

3.3 Edge Computing Use Cases

Rahmani et al. [RGN+18] and Nikoloudakis et al. [NPM+16] implemented edge computing systems in the fields of e-Health and Ambient Assisted Living (AAL). The former is a system for health-monitoring of patients in hospitals or homes. Medical personal should be notified if a patient’s status gets worse and moreover they should be able to view real-time information like electrocardiography (heart rate monitoring) data. Patients are equipped with body-area or implanted sensors transmitting their data to a close gateway using various communication protocols, e.g., Bluetooth, Wi-Fi, ZigBee or 6LoWPAN. These gateways perform data aggregation, filtering and dimensionality reduction. Afterwards analyzed data is sent to the cloud for broadcasting and further data analytics. This data is also used for long-term medical studies. Patients move around in the buildings thus the smart gateways have to support device discovery and mobility in order to avoid data loss and service interruptions. The smart gateways are also capable of temporarily storing sensors’ and users’ data and executing local actions. These actions include information streaming to nearby client devices, controlling of medical actuators and sensor network reconfiguration.

The aim of an AAL systems is the support of elderly and individuals with activity limitations to on the one hand increase quality of life and on the other hand to also reduce health and social care costs. In [NPM$^+$16] the AAL application's task is to alert local authorities and nearby volunteers in case a user leaves a certain geographical area around his/her home and is thus declared unsafe. The alerting service uses a fog node equipped with a 5G small-cell Wi-Fi interface in each home and a wearable embedded device worn by the user. The user's distance to the cell is then periodically calculated based on the received signal strength indicator (RSSI) between the wearable device and the cell. As long as the RSSI can be determined the user is considered safe. Otherwise the user's wearable device connects to the cellular network and transmits information about connected cellular base stations to perform location triangulation. Afterwards the nearest local authority and nearby volunteers will be notified about the user's whereabouts. The implementation is composed of five services that are coordinated by an orchestration entity deployed in the cloud. A service logic component forms the central unit and models the above described business logic, and interacts with all other services. A positioning service performs the user's distance calculation. A location-to-service translation service computes the nearest public safety answering point based on the transmitted cellular information. A profiling service encapsulates all stored personal user's information and implements an access control scheme to restrict information for authorities and volunteers. As long as a user stays within the predefined boundaries the architecture ensures that all processing is done locally and no information has to be transferred to the cloud, moreover personal information is protected from unauthorized access. Experiments showed that the time until a response from either an authority or a volunteer is below five seconds on average.

Both articles showcased the viability of the edge computing paradigm applied in the Smart Home. But instead of building the applications on top of a generalized edge framework they implemented a bespoke fog architecture especially tailored to their requirements. So the main difference between this thesis and these use case evaluations is the usage of a common edge analytics platform that provides the low level tools required for the edge computing paradigm and enables developers to focus on the analytics logic itself. Furthermore their evaluation is mostly based on the usability of the final product rather directly comparing an edge computing approach to a cloud native implementation. In contrast thereof this thesis combines all of the introduced related work by evaluating a Smart Home use case implemented with a serverless edge analytics platform.

CHAPTER 4

Use Case: Elastic Heat

To examine the serverless computation model in the edge computing paradigm we define a Smart Home use case that is composed of typical IoT components so that evaluation results can be universalized across IoT smart home applications and requires computation and storage at the edge. First, we outline the implemented use case, called *Elastic Heat*, in this chapter. At the beginning we explain the initial situation and consequent shortcomings of a conventional water heating system. Then we present ideas to improve and resolve the identified drawbacks. Subsequently anticipated benefits are listed that will also be explored in Chapter 6.

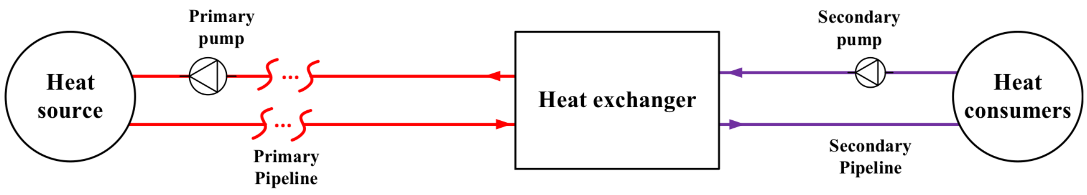


Figure 4.1: Conventional district heating [WSLW15]

The use case we implement throughout this thesis is about household water heating systems, more precisely about drinkable water heating used for tap water and showering. In Austria 21% of yearly heat generation are produced by district heating cogeneration plants, according to a survey of the Austrian Ministry of Economy [Bun17]. Moreover district heating gains popularity and increases every year. The schematic of a conventional district heating system is shown in Figure 4.1. Usually the heat source is a cogeneration plant that supplies households connected to the primary pipeline in a particular area. Each household is equipped with a heat exchanger and a secondary pipeline that in turn is connected to internal heating systems like boilers, radiators or underfloor heating systems. The heat exchanger is the central control unit that ensures that each of these systems has exactly the temperature as configured.

4.1 Problems

In case of a boiler the heat exchanger acts on a predefined plan to check its temperature if it is below specified threshold and starts heating accordingly until a fixed setpoint is reached. This almost static configuration in combination with mostly irregular water consumption raises several problems. The imaginable extremes with this configuration possibilities are to check the temperature either only once a day or on the contrary all day long. In the former case it is very likely to run out of warm water before the end of the day despite additional warm water may be needed. This is because on the one hand a boiler can store only a limited amount of warm water at a time and on the other hand the temperature of the water inside is also capped to increase boiler lifetime. In the latter case the boiler constantly heats up after losing some thermal energy and so maintains its temperature even through nights when almost no warm water is needed. Additionally a boiler usually loses temperature to the surrounding environment despite no water is drained. At our test object we measured a loss of 9.6°C per 24 hours. Therefore to cover the daily water consumption, minimize constant heating and temperature losses usually a setting inbetween these two extremes is desired, but irregular activities like taking a bath hamper finding the best scheduling.

Additionally a trend towards sustainable energy systems in the private sector has emerged. Many residential buildings and homes are equipped with a photovoltaic system (PVS) or solar panels. But despite the eminently overall positive ecological impact some downsides exist that can be further improved. On sunny days a PVS generates more power than a home can consume and thus must feed the surplus into the electricity grid. However power providers normally enforce an upper limit on power fed into the electricity grid, therefore exceeding energy is wasted. So to prevent this behavior rechargeable batteries can be used to accumulate surplus energy or the energy can additionally be used to heat water in order to preserve otherwise lost energy. Yet again this is constrained by the battery capacities and a tradeoff between temperature and attrition rate: the higher the boiler temperature the higher the attrition due to e.g., calcification. Especially in combination with the static configuration of a heat exchanger, that is not capable of responding to external circumstances or a secondary heat source, the boiler has often already reached the desirable temperature, hence may block heating by surplus energy.

4.2 Proposed Solution

Driven by the house owners' desire to minimize their energy costs we propose a system that replaces the static time-based scheduling by a dynamic consumption-oriented scheduling called *Elastic Heat*. Moreover the proposed system should be capable of energy minimization fed into the electricity grid and thus improve sustainable energy efficiency. The fundamental concept of our approach is based on two principles:

1. Heat only when warm water is consumed

2. Heat only as much as needed for consumption

So instead of keeping the boiler at a certain temperature throughout the day, the system should be aware of when and how much water is needed. Using this knowledge we are able to schedule the next heating treatment accordingly and thus minimize heating when warm water is actually consumed. Furthermore the power used to heat the water is also drastically reduced as the boiler only heats exactly the right amount of water, instead of heating until reaching a fixed setpoint. As a consequence the average boiler temperature is lowered, thus it is likely to cut overall energy / costs for water heating. Lower average boiler temperature also results in less temperature loss and above all ensures that water can be heated more frequently by the surplus energy of the PVS. This in turn decreases energy fed into the global electricity grid and further facilitates energy-self-sufficiency. Proper scheduling also means that heating is delayed as much as possible while guaranteeing sufficient warm water for inhabitants. This delay ensures that the PVS is more likely to provide the energy to heat the water, because some scheduled heatings can be skipped if the PVS has already heated the water sufficiently in the meantime.

Figure 4.2 depicts the basic structure of the system's hardware so that the above-mentioned concept can be implemented. In addition to the essential district heating system components, the PVS adds photovoltaic panels on the roof, an inverter that performs DC-to-AC conversion and also implements energy management and a heating cartridge mounted at the boiler. Our testing site additionally has a rechargeable battery to temporarily store surplus energy. In order to implement both optimization principles the house residents' warm water usage behavior has to be known in advance. Therefore we equip the home appliances with IoT devices that provide sensing and actuation. These devices should provide the necessary information, e.g., boiler temperature and warm water consumption, that enables our consumption-oriented scheduler. The system then applies machine learning techniques on this gathered historic data to model and predict future consumption behavior to schedule necessary heating treatments. In addition real-time analytics is utilized to take necessary actions on time. Both results in conjunction enable the prediction of the next necessary water heating treatment, the specific amount needed at that time and moreover prevent non-essential heat up of water.

The heating system is one of the residencies' core systems that provide essential services for its inhabitants. For this reason any management application has to be overly resilient against external influences. Our specified IoT application has to handle limited internet connectivity, slow bandwidth or entire connection outages but also internet service failures. In addition missing measurements from sensors may downgrade forecasting capabilities. Specifically missing real-time data from sensors may completely prevent accurate scheduling. Besides these technical challenges such a continuous monitoring system also faces ethical obstacles. Complete logging of water usage behavior and accompanying information contain highly private data that have to be protected. However providing personal consumption forecasting prevents data anonymization to a certain

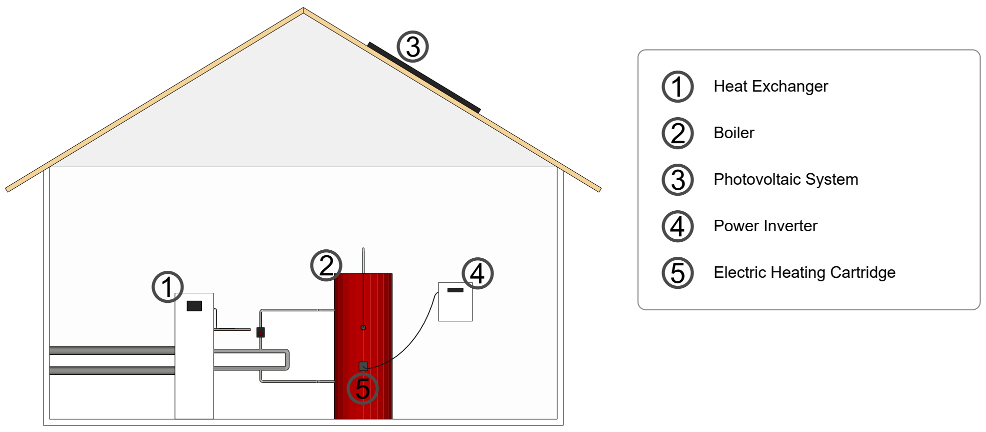


Figure 4.2: Home equipped with district heating and photovoltaic system

degree. Because of these unique challenges the presented use case is a prime example for edge computing. Extending the cloud to the edge facilitates high service availability despite limited internet connectivity or slow bandwidth. Furthermore having a central edge gateway that acts as local datastore at the same time also prevents service failures during connection or cloud service outages. It also minimizes data privacy issues because sensitive information is kept locally, whereas globally shared data can be completely anonymized.

CHAPTER 5

Design

The use case we presented in the previous chapter contains some unique challenges for supporting systems. High availability and resilience against external factors are crucial for non-stop operation. Moreover these systems process substantial private data about inhabitants' preferences and actions. Special attention has to be paid to prevent unintentional data disclosure. In this chapter we present our design to address these challenges. We use one of the recent edge computing frameworks leveraging a serverless architecture. For that reason we have a look at the variety of existing frameworks and perform a brief evaluation. Afterwards we proceed with the description of the basic architecture driven by the selected framework on the one hand and certainly by the use case itself on the other hand. Having drafted the components of our system we continue with the creation of our machine learning model. In this section we encompass the necessary data we have to gather to be able to make a reasonably accurate prediction of water consumption in the next time period. Eventually we illustrate the consumption-oriented scheduling approach that forms the core system component.

5.1 Edge Computing Framework

The emergence of the edge computing paradigm in the form of cloudlets [SBCD09] and fog nodes [BMZA12] has fostered the development of frameworks that promote analytics at the edge. There exist many providers offering services in the field of IoT device fleet management and data processing, but only a few adopted their products to enable computing at the edge in close proximity to the devices. In this context we focus on the most elaborated frameworks namely AWS Greengrass [Amaa] and Microsoft Azure IoT Edge [Micb]. IBM Watson IoT Platform Edge [IBM] is listed for the sake of completeness, but because it is still in limited preview at the time of writing and thus lacks a lot of online resources and information it is not discussed in detail. Other IoT service providers exist that also have a device gateway in place, but only for the purpose of integrating

	AWS Greengrass	Microsoft Azure IoT Edge	IBM Watson IoT Platform Edge
Release Date	June 2017	May 2017	March 2018 (Preview)
Business Logic Deployment Model	Serverless Functions using AWS Lambda	Docker Containers + Azure Services	Docker Containers + Edge Services
Function/Module SDKs	Java 8, Python 2.7, Node.js 6.10	C, C#, Java, Node.js, Python	–
Data Routing Mechanism	MQTT topic subscriptions	Subset of IoT Hub query language	–
Device SDKs	Embedded C, JavaScript, Arduino Yún, Java, Python, iOS, Android, C++	C, C#, Java, Node.js, Python, iOS	–
Device Security Model	TLS with X.509 certificates	TLS with symmetric keys + X.509 certificates	TLS with token authentication + X.509 certificates

Table 5.1: Comparison of Edge Computing Frameworks

things using communication technologies other than ethernet. The frameworks from AWS, Microsoft and IBM allow the deployment of application specific custom code on the edge gateway. AWS Greengrass fully adopt the serverless architecture for the edge gateway, whereas Microsoft Azure IoT Edge and IBM Watson IoT Platform Edge rely on containerization of edge modules/services. Table 5.1 summarizes the key differences between the frameworks which we discuss in this section.

The basic building block of these frameworks is the concept of a gateway deployed at the edge whose main task is to facilitate device connectivity through a configurable message broker by means of data routing mechanisms. Furthermore the frameworks provide a way to remotely deploy functions or modules via a platform hosted in the cloud. The arrangement and the interaction of different functions/modules are controlled by the message broker by routing messages inbetween these components. Another feature these frameworks have in common is the support for device shadows also called device twins.

A shadow is a virtual copy of a device's state and thus abstracts the actual sensor or actuator using persistent message queues. Basically each IoT thing subscribes to its shadow topic and listens for published messages. The persistent queue allows devices to become offline without affecting other communicating things. As soon as it is online again it receives the latest shadow state pushed to the queue. This has the advantage that functions and IoT devices are highly decoupled. A software developer does not have to care about the receiving device's state. Instead the message broker takes care of retransmitting data when the device comes online again.

In the leading paragraph we talked about aspects the frameworks share, now we focus on things they handle differently. The first and most important aspect we have a look at is their deployment process of custom business logic. AWS Greengrass leverages the AWS Lambda [Amac] service. It is possible to create an AWS Lambda function, upload desired code and deploy it to the edge gateway - the Greengrass Core. The core takes care of provisioning and spawning the runtime environment for the functions. At the time of writing runtime environments for Java 8, Python 2.7 and Node.JS are supported. Microsoft Azure IoT Edge uses Docker containers as deployment units on their gateway devices. The framework itself is also shipped in multiple Docker containers. Besides the core software Microsoft Azure IoT Edge further provides certain dockerized Azure services like Azure Stream Analytics [Micc] or Azure Functions [Mica] to be used as modules. Custom modules can be implemented in C, C#, Java, Node.JS and Python.

The composition of all deployed functions/modules to the actual business logic is done by the message broker defined by certain routing mechanisms. In AWS Greengrass these routes can be defined by so called subscriptions which have a source and a target endpoint. Furthermore these subscriptions can be filtered upon MQTT topics. Besides reserved predefined topics for amongst other things device shadows, topics can be arbitrarily chosen. Microsoft Azure IoT Edge uses a subset of the IoT Hub query language for the purpose of data routing. They provide similar functionality but additionally support data filtering.

The central part of IoT applications form the actual things — sensors and actuators. Support for various programming languages is thus essential to such frameworks. The AWS Greengrass underlying AWS IoT service has support for Embedded C, JavaScript, Arduino Yún, Java, Python, iOS, Android and C++. So far only SDKs in C++ and Python have been updated to directly support AWS Greengrass. The update comprises a feature called Greengrass Discovery Service required to retrieve host information and certificates of the Greengrass Core. So when using the other SDKs someone has to implement the Greengrass Discovery by oneself, utilizing the Greengrass Discovery RESTful API. In contrast Microsoft Azure IoT Edge provides SDKs for C, C#, Java, Node.JS and Python.

The things in IoT are typically placed in untrusted environments, which opens another attack vector for intruders. To mitigate such attempts the IoT system acts on a security model. This model specifies how parts of the system interact with each other, how they can authenticate to further be authorized for certain actions. In AWS Greengrass each device has a unique X.509 certificate which is used to establish a TLS connection between the different devices, the Core, or the IoT cloud. Furthermore each certificate has one or more policies attached. These policies authorize the certificate holder for certain actions on certain resources. The same hold partially true for Microsoft Azure IoT Edge. Here policies are called permissions and besides support for X.509 device certificates, the service also supports TLS connections with symmetric keys.

In summary, both Greengrass and Azure IoT consist of almost identical building blocks and provide essentially the same features. The most fundamental divergence between them is the deployment model. AWS Greengrass fully leverages a serverless programming model, whereas Azure IoT Edge uses containers. This implies that Azure IoT Edge grants more freedom to the developers since they are not tied to any runtime environment as it is the case with serverless functions. On the contrary building containers is more time-consuming and requires domain knowledge. Besides the deployment model, frameworks also have to provide SDKs for functions/modules to interact with edge devices. At the moment Azure IoT edge offers SDKs for more programming languages than AWS Greengrass and also has a more advanced data routing mechanism, supporting routing-decisions based on transmitted data. In contrast AWS Greengrass implements more IoT device SDKs, enabling more diverse things in different languages to connect to the service. But these differences are only nuances and basically the decision streamlines to familiarity of development teams with a cloud providers and quite likely the existence of in-use cloud infrastructure. Because AWS Greengrass fully incorporates the serverless architecture we implement our *Elastic Heat* use case, introduced in Chapter 4, using AWS Greengrass. However, it has to be stated that Greengrass and Azure IoT share essential architectural aspects and components. For this reason the decision for a certain framework does not significantly influence the fundamental design of our system. Moreover, the design is generalizable and can be applied to any serverless edge platform.

5.2 Architecture

Our application's meta-architecture is predetermined by the AWS Greengrass framework which combines IoT devices to so called Greengrass Groups (GGGs). Each group encapsulates the configuration of a Greengrass Core (GGC) device and its associated IoT devices that are interconnected via a local area network. A reasonable architecture for serverless predictive analytics at the edge in Smart Homes is depicted in Figure 5.1. The illustrated architecture represents an individual GGG. Mapped functions build the predictive analytics system's core components. These consist of one or more functions that are responsible for persisting sensor data and perform initial data processing steps. Additionally a regressor and predictor function perform model training as well as forecasting. Supplementally a scheduler can be used to make decisions based on the forecast and orchestrate IoT devices or additional functions to act accordingly. The scheduler then instructs executor functions to put the planned actions into practice. The depicted cloud segment is globally available to all existing groups. In an edge computing manner IoT devices of a GGG communicate with the locally available GGC instead of directly transmitting data to the cloud. Any connection and transmission of data between devices and the cloud leverages the Message Queueing Telemetry Transport (MQTT) protocol characterized by lightweight and efficient design.

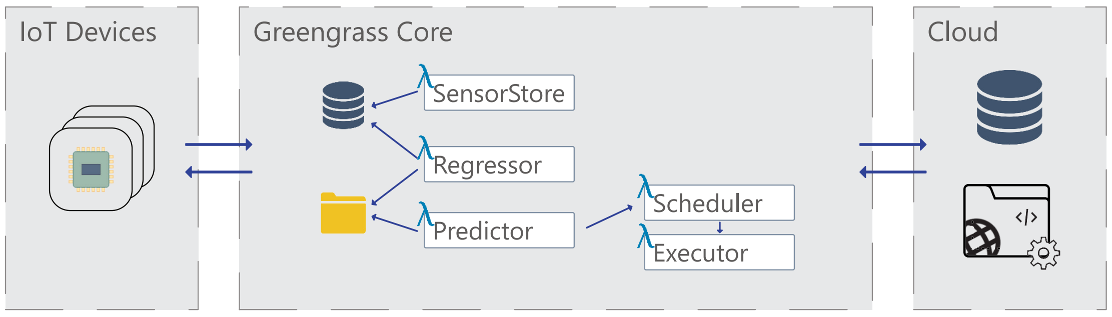


Figure 5.1: Fundamental architecture of a serverless predictive analytics platform at the edge. Implemented in *Elastic Heat*.

5.2.1 Elastic Heat Components

Our *Elastic Heat* system is based on this generalized architecture and comprises all functions from the fundamental predictive analytics architecture. The Greengrass framework provides two types of functions:

- Event-triggered

  Event-based functions are triggered by certain events that occur during the system's runtime. An event can be an IoT message from a particular device, on a specific MQTT topic or directly invoked from within another function running on the GGC.

- Long-living

  These functions are immediately invoked on deployment or startup of the GGC. For this reason they are suitable for continuous workloads and periodic tasks that are independent of events.

The GGC provides all required functionality to remotely deploy these functions with orchestration information for a GGG from a cloud platform. As shown in Figure 5.1 the *Elastic Heat* system is composed of five functions running on a GGC. Some of these connect to a local data storage, specifically a locally running database and the local filesystem. Any function executed on a GGC can request real-time data from associated IoT devices, invoke other local functions and also forward data to the AWS IoT cloud service as long as it is allowed by the subscriptions in the GGG orchestration information.

SensorStore

The first function, the SensorStore, is responsible for persisting data from the sensors required for the machine learning model. All sensory data that belongs to the same machine learning model instance have to share the same timestamp in order to be identified as an instance in our Regressor function. For that reason we designed a long-living function to actively collect the data from the various devices in a specified interval instead of letting the function be triggered on a shadow delta event. Furthermore, this has the advantage that the time interval between the measurements is centrally configurable. The data is gathered by using the shadow of each involved IoT device. At the same time the devices independently update their shadows as soon as the device's state changes. Before the data is persisted, the SensorStore function applies various data aggregation and filtering methods. An example thereof is the conversion of units: from total volume in liters to liters per hour. After data preprocessing the function persists the data in a locally deployed relational database.

Regressor

The Regressor function accesses the previously stored data and further prepares the data to generate the machine learning model used for water consumption forecasting. Most notably it has to extract the necessary features for our regression model. This step also includes extrapolation of date information, e.g., whether it is an official holiday, or what day of the week it is. Another important phase is the unwinding of circular features and creating the optimal feature lags. We explain these preparation steps in more detail in Section 5.3. After applying all these steps the Regressor trains and persists the model on the filesystem for later usage by the Predictor. However, each function is executed in an isolated ephemeral environment, so they usually do not share any resources including the filesystem. But it is possible to mount a folder from the host environment into the serverless function's execution environment such that the function can read and write files in that directory.

Predictor

This function forecasts the water consumption. Every beginning of an hour the function loads the persisted regression model from the mounted folder shared with the Regressor. Using this model it predicts the water amount needed in the hour after the next based on the most recent sensed data. Before that it applies the same preparation steps as the Regressor, but only to build the most recent model instance for prediction. If the forecast yields a required heating it invokes the Scheduler function with the predicted amount.

Scheduler

The Scheduler is the core component of the system. Its purpose is to convert the previously predicted amount of water to concrete water temperature in the boiler. At the same time it estimates the boiler temperature at that particular time to decide if heating is even necessary. In the case of a necessary heating the Scheduler then calculates the time needed to heat until the desired temperature is reached. Eventually the function yields the time for the next heating, the duration and the necessary boiler temperature.

Executor

The Executor function executes the planned heating. Using the information provided by the Scheduler this function waits until planned time and commands the heat exchanger to start heating. When the calculated duration is over it sends another message to the heat exchanger to stop heating.

5.2.2 Elastic Heat Process Flow

The interplay between all these components/functions at the consumption-oriented scheduling is shown in Figure 5.2. As can be seen the Regressor and the Predictor are both long-living functions running on the GGC. The Regressor's main function repeats itself daily at midnight to retrain the model including the newly measured data. After every training session it dumps the new model to a certain folder on the filesystem of the GGC that is also accessible by the Predictor function. At the same time the Predictor function repeats itself every beginning of an hour to predict the next future water usage, based on the stored model and real-time data. Afterwards it invokes the Scheduler function and transmits three quantities: the previously predicted volume, the current predicted volume as well as the time when the predicted volume is needed. Based on this data the Scheduler decides whether heating is required to provide the amount of warm water in the first place. If it is the case then the Scheduler further calculates the exact time when to start the heating process and the respective duration. Eventually it invokes the Executor function sending along all the calculated information. Figure 5.3 depicts both scenarios on a timeline. Example 1 illustrates a scenario in which no heating is scheduled because either the prediction is close to zero and/or the boiler temperature is sufficiently warm. The other case, as described above, is represented in Example 2. As shown in Example 2, as soon as the Scheduler invokes the Executor function it starts to wait until the planned starting time. Reaching that time it kicks off the heating process by changing the shadow state of the heat exchanger unit. Then it keeps waiting again until the duration is over. At that moment it changes the shadow state back to stop heating.

5.3 Machine Learning Model

In the previous section we outlined the overall process flow and how we use the machine learning model for the consumption-oriented scheduling approach. In this section we focus on the machine learning model itself. Initially we present the regression algorithm used in our Elastic Heat system. Then we describe how we selected our features for our model and give an in-depth list of them. Afterwards we discuss the preparation steps we apply on our raw data in order to extract the individual features. In a further step we

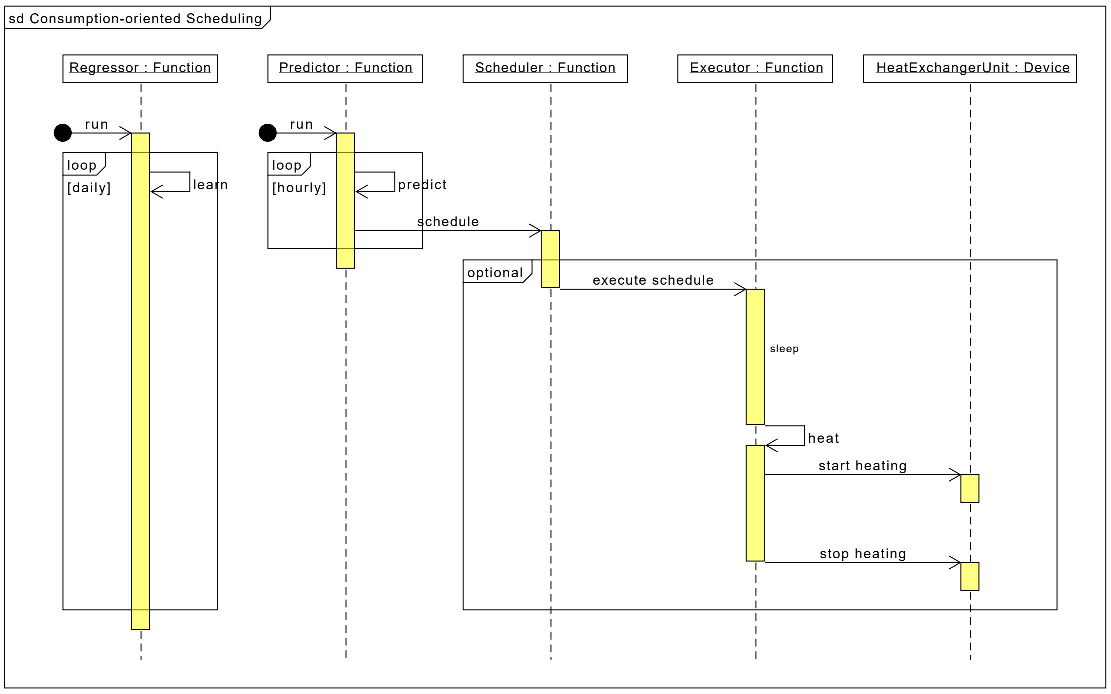


Figure 5.2: Scheduling Process Flow of *Elastic Heat* components/functions

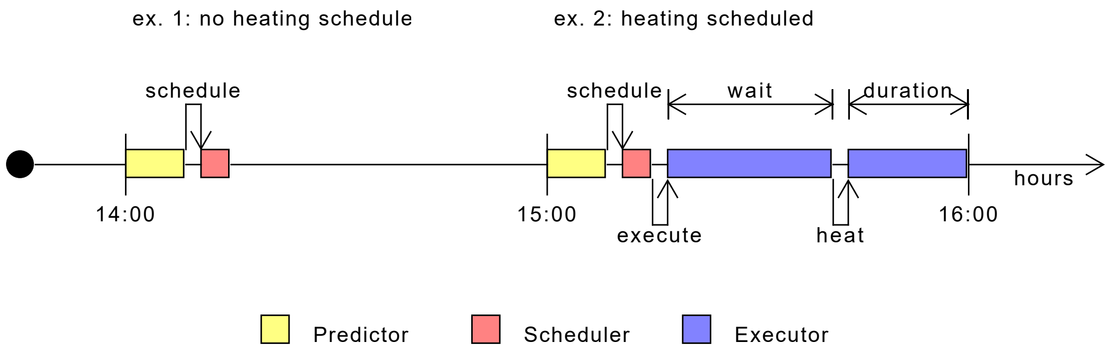


Figure 5.3: Examples of derived schedules and the execution thereof.

evaluate the training time of our model on the GGC, in particular the performance as time passes and the number of instances increases. After validating the model using time series cross-validation we describe the deployment of the machine learning model in our system.

As we have shown in Chapter 4 and the previous Section 5.2 it is necessary, in our system, to predict the volume of water that is consumed from the boiler. Specifically, as illustrated in Figure 5.3, our model should be capable of forecasting the values an hour before actual consumption to have time to heat the water. In the example taken from the figure, the Predictor has to estimate the volume needed from 15:00 to 16:00 at 14:00.

For our use case implementation we choose the random forest regression algorithm [Bre01]. As stated by Breiman random forests have many positive characteristics. They are resistant against overfitting, they give insight into variable importances thus perform implicit feature selection. Furthermore, a comparison of supervised learning algorithms by Caruana R. et al. [CKY08] showed that random forests perform consistently well in multiple different machine learning tasks. Moreover they state that random forests are easy to parallize and efficiently scalable to fit problems with high dimensions. Another benefit is the widespread availability of open-source implementations, with particular focus on the scikit-learn library in Python [PVG$^+$12].

5.3.1 Feature Engineering

As stated by Mozer [Moz05] regularities in Smart Homes are „not only based on time of day and day of week but rather are based on a large number of factors“. With this statement in mind we analyze the factors that may influence warm water consumption, especially while taking a shower or a bath which are the activities with the highest flow rate. Table 5.2 presents a list of external factors that might have an influence on daily water consumption behavior. The first external influence is the outdoor temperature. A lower outdoor temperature may indicate a longer or warmer shower or may even lead to take a bath instead. On the contrary on warmer days someone may take a colder refreshing shower or take a shower instead of a bath, hence need less warm water. A second indicator for warm water consumption is the bathroom temperature. The argument is similar to the outdoor temperature, the colder the more likely it is to have a longer shower or bath. The third aspect in considering water consumption is the actual consumers in the house, in particular the number of occupants. The higher the occupancy level the higher the water consumption. It is clear that four inhabitants drain more warm water than two. Additionally, visitors may increase consumption as well by having, e.g., more dishes to clean on lunch time or because they stay over night and take a shower in the morning or evening. In addition to these factors the research works [APK16, FAS18] state that there are significant differences in consumption profiles between weekdays and weekends. Usually people have a daily rhythm during the workdays, e.g., typical wake-up time or start of work. This rhythm is regularly disrupted on weekends, which may affect the water consumption pattern. Because public holidays normally lead to a non-working day they similarly affect consumption behavior.

In order to measure these external influences from Table 5.2 we add several IoT devices to our test environment. Our test environment is an ordinary single-family home that uses the necessary supporting systems — district heating and PVS. The home equipped with the required IoT devices is shown in Figure 5.4. It depicts all installed sensors at the various locations to observe the factors that affect water consumption behavior. Additionally we equipped the boiler with a volume transmitter which senses the amount of water that is drained and a thermometer to constantly measure the water temperature

Factor	Rationale
Outdoor temperature	In contrast to summer someone may take a longer shower or even tends to take a bath on cold days
Bathroom temperature	If the bathroom is cold it may be cozy in the bathtub or shower tray, thus time spent for washing and water usage are increased
Number of occupants	A high number of occupants may indicate that more people use warm water for washing
Weekday	On weekends the time when someone consumes water may be drastically different compared to workdays
Holiday	A public holiday may influence time and amount of water consumption

Table 5.2: External factors that may influence water consumption behavior

inside. These continuous measurements of the external factors are our independent variables used as features in our machine learning model.

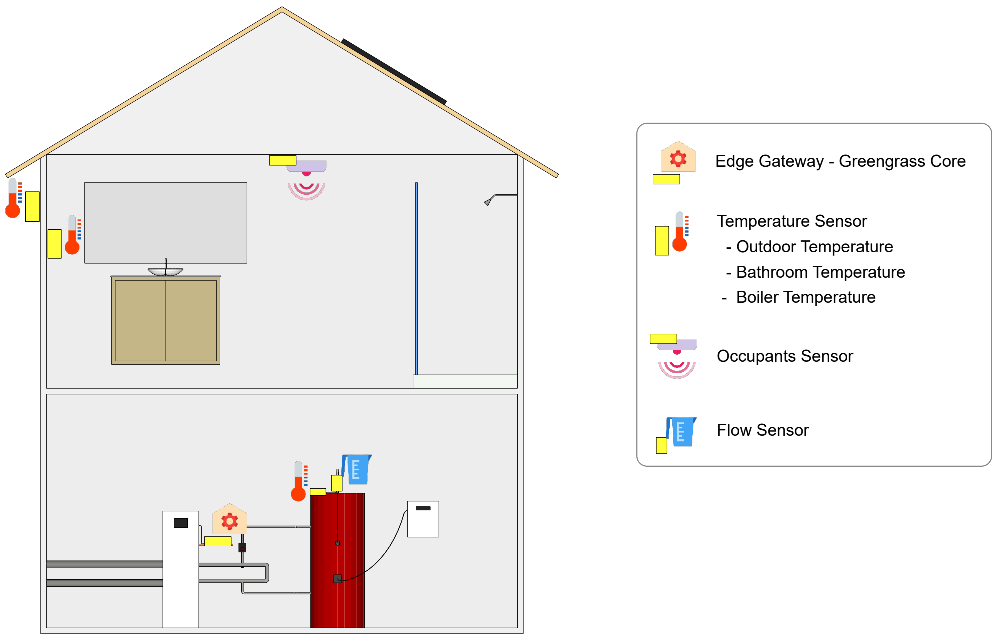


Figure 5.4: IoT devices installed in a home for *Elastic Heat*

Figure 5.5a visualizes a sample from real data collected in our test bed. The sample shows measurements from the outdoor temperature, bathroom temperature, number of occupants and consumed volume in intervals of an hour. The plot reveals a daily seasonal pattern of consumed water that can be traced back to the fact that all residents have

Feature	Description
Hour	The hour when the instance was measured
Day	The day of the month
Weekday	The particular weekday, e.g., Monday, Tuesday, etc.
Week of year	The number of the week of the year
Month	The month when the instance was measured
Holiday	Whether the instance was measured on a public holiday

Table 5.3: Time information used in feature vector

regular work. It has a peak in the early morning and flattens over the day. There is no water consumption at all over night. To further gain additional knowledge of the data's underlying interdependencies we visualize cross-correlation and plot histograms of the different variables in a scatterplot matrix shown in figure 5.5b. The examination reveals a non-trivial relationship between the observed factors and water consumption that cannot easily be recognized with the help of the cross-correlation matrix. In addition to that we can use autocorrelation of water consumption and the correlations to other variables' past values in general. These preceding values are also known as lags or lag values and will be further discussed in the performed preprocessing steps.

To further improve the model we apply several data preprocessing steps and add further values to our feature vector for better accuracy.

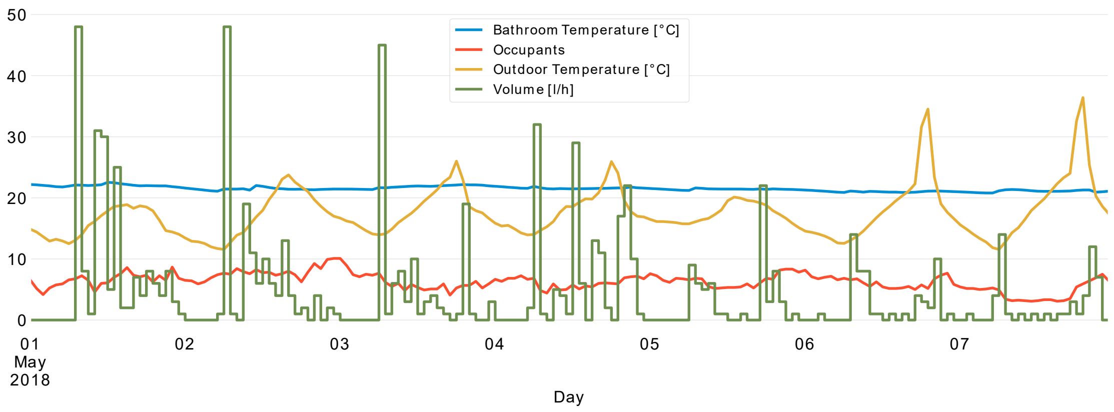
1. Extract additional information from timestamp

   Table 5.3 lists additional features we can extract from the measurement's timestamp. Besides the previously mentioned information about the weekday and public holidays we split up the timestamp into different attributes in a first step. By doing this we have four additional attributes regarding the time. The hour, the day, the week of the year and the month.

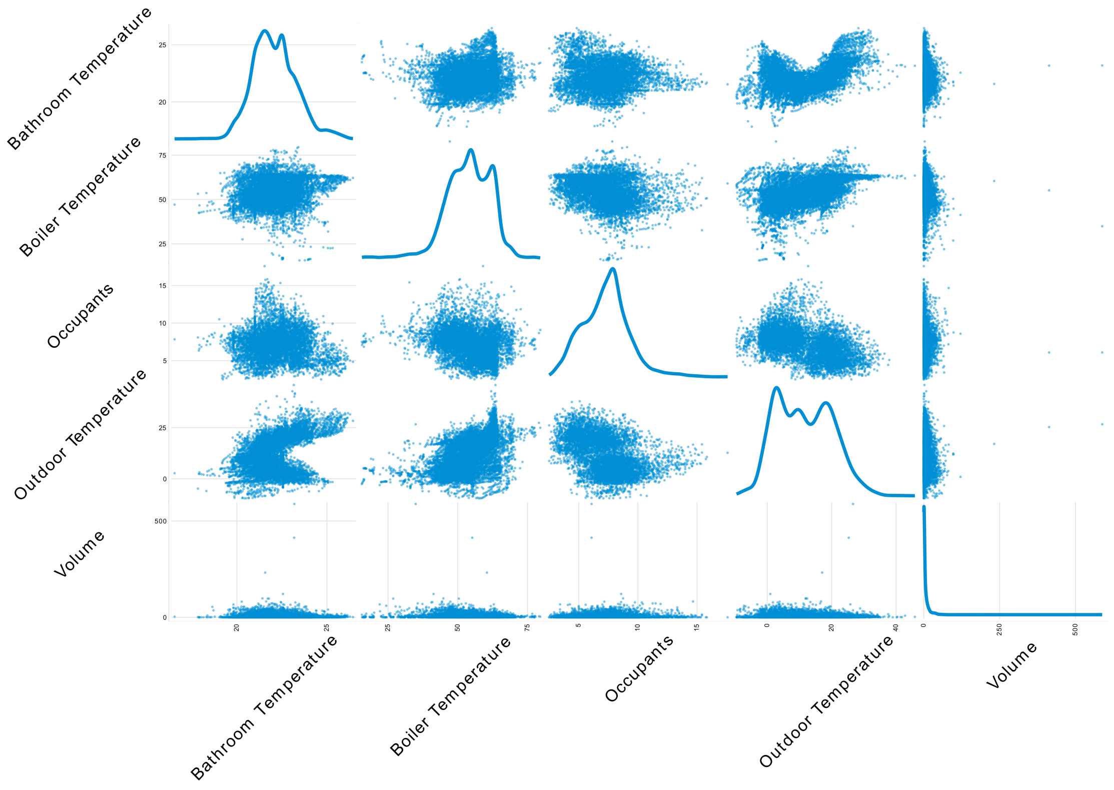
2. Cyclical feature unwinding

   However all of the extracted time-related values are called cyclical features, according to Dossman in [Dos17]. These are features that regularly repeat themselves. By applying machine learning algorithms some information about these get lost. For example, the interval between 23:00 and 00:00 is equal to the interval between 01:00 and 02:00. The algorithm however treats them as pure integers values and thus discards this information. In a referenced blog entry [Kal17] cyclical feature engineering is described in detail. This write-up describes the process that is based on the even distribution of values on the unit circle and thus preserves the equal intervals. The result of this preprocessing step is a new sine and cosine value that replaces the original feature.

3. Shift volume value



(a) Excerpt of gathered sensor data in intervals of an hour.



(b) Scatterplot matrix of the full data set.

Figure 5.5: Different visualizations of sensor data.

By the way the mechanical flow sensor that is built into the input pipe of the boiler operates we can not directly read the water flow values. The main reason for this is because the mechanical sensor reports flow data in real-time to the heat exchanger unit. We would need to continuously send requests to the heat exchanger unit in order to not miss out any values. Any missing value results in a wrong flow measurement. However this would flood the local network with a huge number of HTTP requests. For that reason we instead measure the total amount of consumed water in liters and subtract the last value to calculate the actual amount. A measurement thus contains the liters consumed since the previous one. But because we require the amount that will be consumed in the next interval we need to shift the volume column by one row. This results in a feature vector that contains the liters that will be consumed until next measurement. This shift is extremely important because otherwise we would forecast a volume from the previous interval.

4. Create feature lags

   The last preprocessing step is to create certain lag values for the variables and drop unnecessary ones. So at first we have to figure out useful lag values that improve the overall accuracy of the model. The random forest algorithm has the capability to calculate feature importances of the provided feature set. We can leverage this feature to determine the most important features and discard the rest. For this we expand our data set with lag values for all variables and fit the algorithm with our preprocessed data. The calculated importances are shown by the blue bars in Figure 5.6.

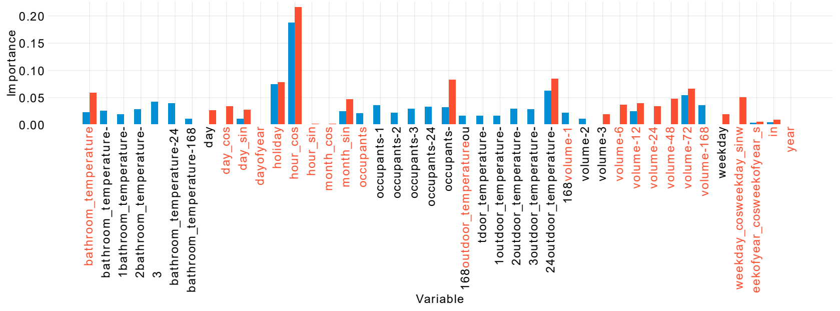


Figure 5.6: Feature importances of random forest algorithm.

   Besides this estimated importances we also calculate the autocorrelation of the measured volume consumption to find suitable past values. Figure 5.7 presents the autocorrelation between the different lag values. This chart unambiguously show a high correlation between the current volume and any interval of 24 hours ahead. It also reveals an interesting negative correlation to six hours earlier and also a repetition every 12 hours. In addition to these two results we also use a trial-and-error method to further increase the accuracy.

After performing all the preprocessing steps above we end up with the final feature vector listed in Table 5.4. Experiments with feature selection techniques showed that adding other variables' lag values, like earlier bathroom temperature or previous number

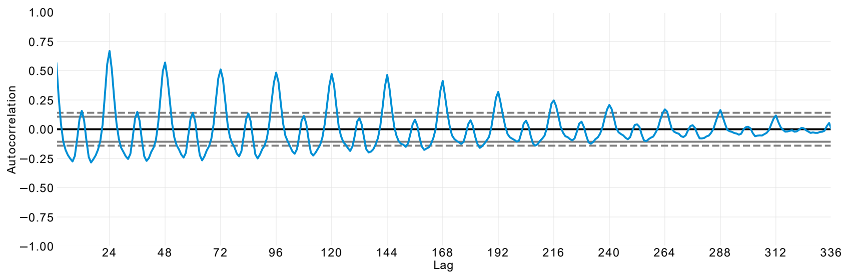


Figure 5.7: Autocorrelation of volume.

Feature	Description
volume_temp-1	Volume consumption the hour before
volume_temp-6	Volume consumption 6 hours before
volume_temp-12	Volume consumption 12 hours before
volume_temp-24	Volume consumption the day before
volume_temp-48	Volume consumption two days before
volume_temp-72	Volume consumption three days before
volume_temp-168	Volume consumption the week before
outdoor_temp	Outdoor temperature
bathroom_temp	Bathroom temperature
occupants	Number of occupants (estimated by connected WiFi devices)
hour_sin	Hour sine part
hour_cos	Hour cosine part
day_sin	Day of month sine part
day_cos	Day of month cosine part
week_sin	Week of year sine part
week_cos	Week of year cosine part
month_sin	Month sine part
month_cos	Month cosine part
holiday	0 or 1 whether it is a holiday

Table 5.4: Final feature vector

of occupants in the residency, reduced the accuracy of the model. For this reason we removed all lags except the most prevalent autocorrelated volume values.

This features inevitably result in following IoT sensors attached to the various components of the district heating system:

- Flow Sensor
  Attached to the outlet pipe of the boiler. Measures the passed through water in volume per hour.

- Boiler Temperature Sensor
  Leverages the inboard temperature sensor of the boiler.

- Bathroom Sensor
  Temperature sensor mounted in the bathroom.

- Outdoor Temperature Sensor
  Temperature sensor mounted on the outside of the house.

- Occupants Sensor
  Estimates occupants by counting connected WiFi devices.

Including these the *Elastic Heat* system comprises of the IoT devices shown in Figure 5.8 depicting a global system overview.

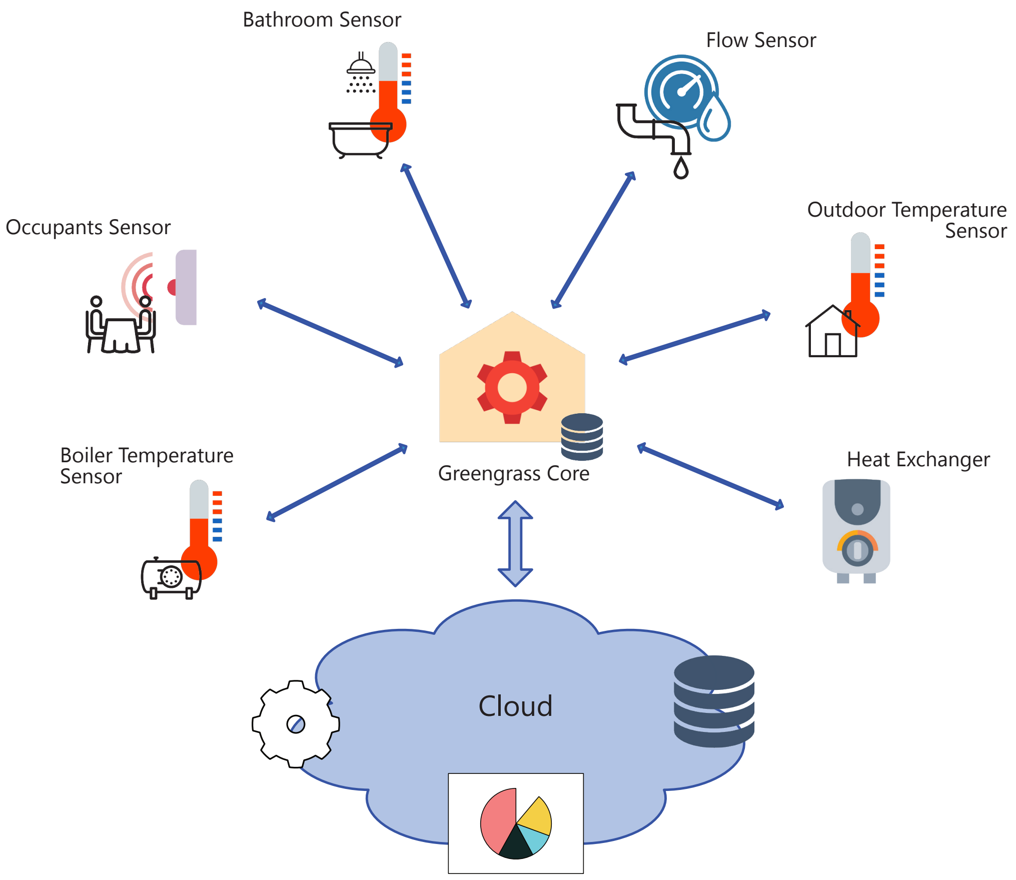


Figure 5.8: Global view of IoT devices, the GGC and the cloud in *Elastic Heat*

hour	bath_tp	out_tp	...	vol
10	21.12	7.08	...	7.0
11	21.10	6.92	...	1.0
12	21.01	6.17	...	6.0
13	20.91	8.08	...	4.36

$\Longrightarrow$

hour	bath_tp	out_tp	...	vol
10	21.12	7.08	...	1.0
11	21.10	6.92	...	6.0
12	21.01	6.17	...	4.36
13	20.91	8.08	...	

Table 5.5: Label set shift on instances for model training.

5.3.2 Training

With the feature set complete we can now proceed to implement the training of our random forest regression model. Because IoT edge devices are typically resource restricted, as mentioned in Section 2.3, we have to analyze expected training duration with respect to the training dataset size. However beforehand we have to further prepare the training data slightly.

Because our goal is to predict the consumption of the full hour after next (see Section 5.2.2), we need to shift our label set by one row once more. This produces instances that contain the volume measurement from the intended timespan. This shift is illustrated in Table 5.5. It can be taken note that this also results in an instance that does not contain any value to train with, therefore we continue to remove any instance that has at least one empty value in any of its columns.

The random forest algorithm is resistant against overfitting, thus too many variables do not negatively affect the accuracy of the regression model. However the more dependent variables the longer the training period of the model. This and additionally restricted processing power of edge devices logically get us to decrease the instance size for better scalability at the edge. Figure 5.9 clearly exemplifies the performance gain through visualization of training time with respect to the sample size — once executed with all lag values (blue line) and once with the final reduced variables (orange line). Both show a projection for the amount of data collected over five years ahead. According to these experiments the training, using the bigger feature vector, would take about an hour with approximately 43.000 samples — the amount after five years. In contrast the simpler feature set requires only a third of that time, in particular roughly 20 minutes after five years. Taking this results into account it is feasible to retrain our model daily for many years before reaching the limit of resources.

5.3.3 Validation

The accuracy of our model directly influences the effectiveness of our scheduling approach, consequently our whole smart heating system. Therefore we define the accepted discrepancy between the actual required volume and the prediction to be less than three percent. The reason for this threshold is that a three percent error results in a difference from merely about half a degree boiler temperature at our testing site, equipped with a

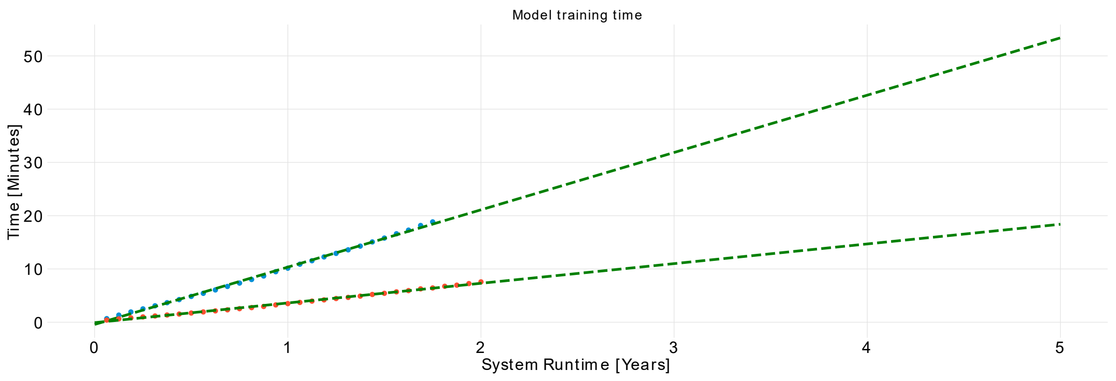


Figure 5.9: Estimation of training-time to system runtime.

200 liter capacity boiler. This in turn guarantees that we can simply heat half a degree more than needed without wasting to much energy. A three percent error implies that at a total capacity of 200 liters the error has to be less than 6 liters.

In order to evaluate the accuracy of our model we perform a time series cross-validation on our latest refined model. Time series cross-validation splits the data into N chunks. Figure 5.10 illustrates this process using a rolling window approach. As can be seen in the figure, on each iteration the window is moved to the right by the size of one chunk. Data to the left of this junk, therefore earlier data, is then used to predict the hold back data within the window. Training data is visualized in blue, whereas the red color highlights data that has been predicted. The line colored green shows the original data, used to calculate the mean absolute error.

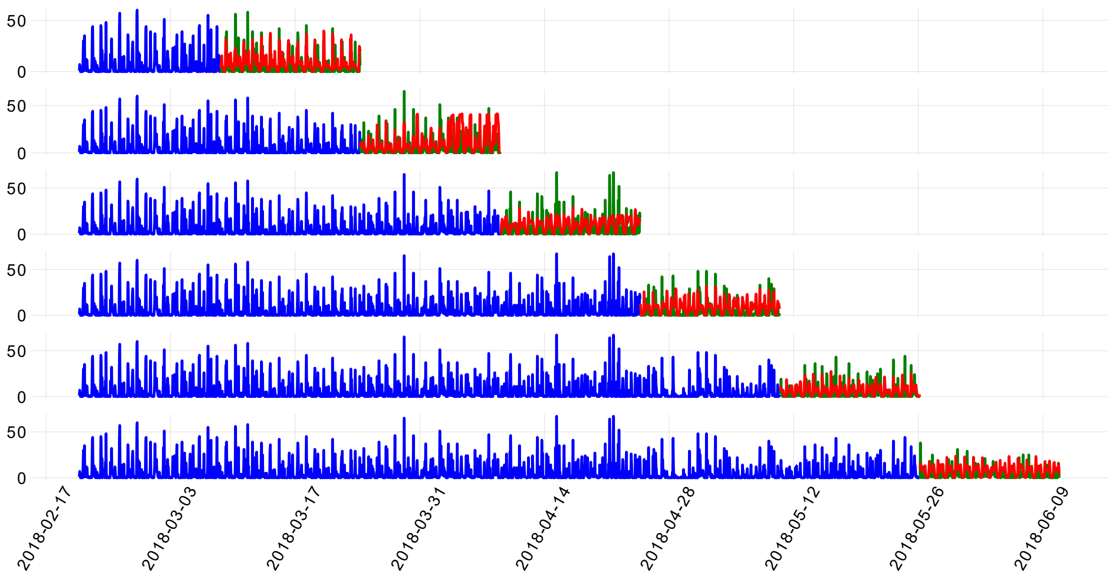


Figure 5.10: Time series cross-validation of model using six splits.

The mean absolute error calculated by the cross-validation method is 6.4l/h, thus slightly bigger than our set requirement. Nonetheless it can be noted that the random forest regression model successfully models the repetitiveness of the consumption behavior to a certain degree. However apparently the model has some problems predicting the peak values. So to assure enough warm water we constantly add this estimated error to the prediction in our scheduling approach, since it only results in heating half a degree more.

5.4 Consumption-oriented Scheduling

The scheduling approach implemented in our use case is entirely based on the consumption behavior of inhabitants. It hourly forecasts the consumption rate two hours ahead, as explained in Section 5.2.2. After the forecast it has to convert the predicted amount from liter per hour to a concrete boiler temperature that provides the required volume. Then it has to calculate the time to start heating. This heating duration is also significantly affected by the initial boiler temperature. Since we start scheduling an hour before actual water usage this temperature is going to change because of water consumption in the meantime. Therefore we also need to estimate the initial boiler temperature for the beginning of the next hour by combining a real-time temperature measurement with the predicted amount for that particular period.

The Algorithm 5.1 describes the basic functionality of the Scheduler lambda function. It takes the previous predicted volume and the current predicted volume to estimate the heating duration. The READ__TEMPERATURE__FROM__SENSOR function directly retrieves the temperature from the boiler temperature sensor. We discuss the CALCULATE__ functions in more detail because they represent the essential calculations mentioned in the previous paragraph.

Algorithm 5.1: Scheduling based on volume prediction

```
Input: The previous predicted volume $\kappa$ and the current predicted volume $\epsilon$
Output: Heating duration
1 $T_{boiler}$ = READ__TEMPERATURE__FROM__SENSOR() ;
2 $T_{anticipated}$ = CALCULATE__TEMPERATURE__AFTER__CONSUMPTION($\kappa, T_{boiler}$) ;
3 $T_{desired}$ = CALCULATE__DESIRED__BOILER__TEMPERATURE($\epsilon$) ;
4 if $T_{anticipated} \geq T_{desired}$ then
5     return $0, 0$ ;
6 end
7 $t_{heating}$ = CALCULATE__REQUIRED__TIME__FOR__HEATING($T_{anticipated}, T_{desired}$) ;
8 return $t_{heating}$ ;
```

5.4.1 CALCULATE__TEMPERATURE__AFTER__CONSUMPTION

This function determines the anticipated boiler temperature at the beginning of the prediction's timespan. This temperature, in turn, is required to calculate the required heating time more accurately. We use Richmann's calorimetric mixing formula [Ric76] that simplifies to a weighted average when used with equal materials, e.g., cold and warm water. It is possible to use this formula because consuming warm water is the same as replacing the hot water by the same amount of cold water in the boiler. So the rest of the warm water mixes with the cold water until an equilibrium is found. It is important to note that an equilibrium between the two is not established immediately. But since we want to estimate the temperature that is approached after an hour this should accurately reflect the true value.

The formula is given in Equation 5.1.

$$T_m = \frac{m_h T_h + m_c T_c}{m_h + m_c} \tag{5.1}$$

The result T_m represents the temperature of the mixed water. It is the result of mixing T_h warm water of m_h mass with T_c cold water of m_c mass. In our use case T_c is a constant of roughly $10°C$ and $m_h + m_c$ equals the boiler capacity. The general case of this equation expects masses, e.g., kilogram, to be inserted, however since we use it for mixing materials of equal mass density we can insert our volume information as well. Furthermore, the formula is only valid with the temperatures given in units of Kelvin.

5.4.2 CALCULATE__DESIRED__BOILER__TEMPERATURE

We could rearrange Richmann's formula such that it depends on the amount of warm water replaced by the cold water and thus calculates the required hot water temperature T_h

$$T_h(m_c) = \frac{T_m(m_h + m_c) - m_c T_c}{m_h}. \tag{5.2}$$

Because the boiler should reach the lowest temperature that is still perceived as warm after consuming water, we define

$$T_m = 311.15K \ (\widehat{=}\ 38°C) \tag{5.3}$$

for this rearranged Equation 5.2. However this equation does not accurately describe the dynamic behavior of the boiler's water temperature.

In fact m_h is dependent on m_c by

$$m_h(m_c) = 200 - m_c \tag{5.4}$$

because of the boiler's capacity. The resulting effect is shown in Figure 5.11. The graph shows the timespan between usage start and the point in time as soon as the water was perceived too cold for shower usage ($\leq T_m$). The orange dots represent the measured boiler temperature. In total we drained the full boiler capacity of 200 liters until then. Furthermore, the graph reveals that Equation 5.2 approaches infinity with higher hot water amounts m_h. Therefore we cannot use this equation to calculate the required temperature for a certain amount of warm water.

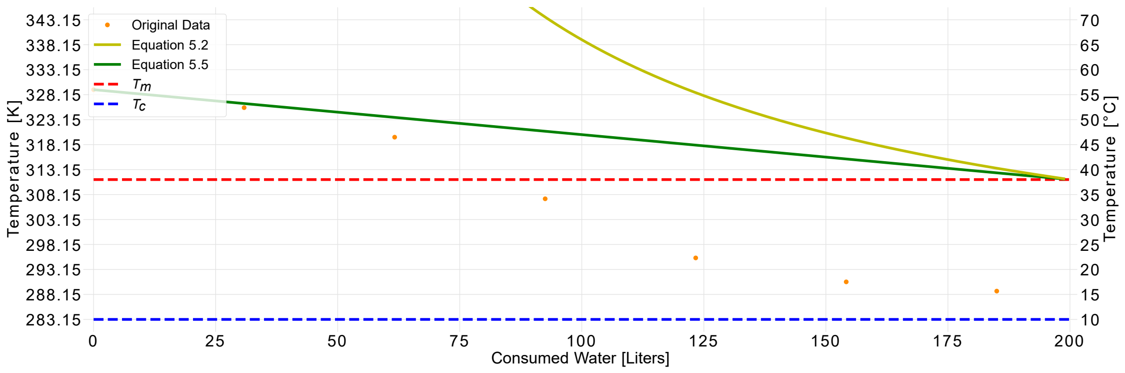


Figure 5.11: Boiler temperature when consuming warm water

The graph also shows that the boiler's water temperature was roughly $17.5°C$ while the drained water was still perceived warm. This effect can be explained by Figure 5.12. Before actual warm water consumption the boiler is fully heated at a particular temperature, as shown in Figure 5.12a. While hot water is drained from the top when consumed, cold water is pumped into the boiler from underneath, Figure 5.12b. But because the consumption happens faster than the mixing of the differently tempered volumes of water, the warm water on top has only partially cooled down at this point. Moreover, because the boiler's temperature sensor is placed centrally, it reports a colder temperature even if there is some more warm water left on top. This results in practice, in having more warm water available than measured.

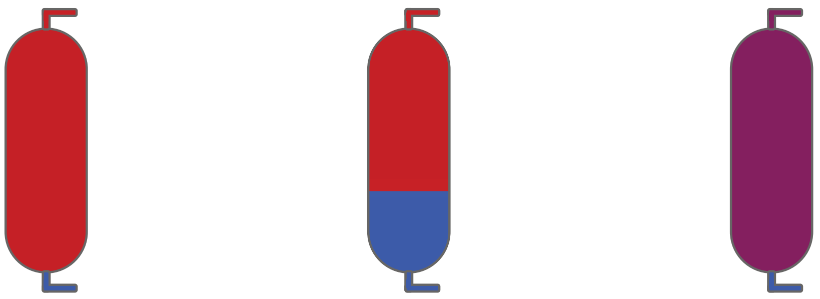
Nonetheless we can use Equation 5.1 to calculate the temperature after water consumption in Section 5.4.1. Since we are interested in the boiler's temperature after some time the water temperature has already found an equilibrium until then according to Richmann's calorimetric mixing formula. This equilibrium is illustrated in Figure 5.12c, which shows a uniform boiler's temperature.

To derive a more accurate function that calculates the desired boiler temperature we plot the perceived temperature curve by linearly interpolating the initial temperature with the point of $38°C$ at 200 liters.

$$T_h(m_c) = -\frac{18}{200}m_c + 329.15 \tag{5.5}$$

With the resulting interpolated function 5.5 and because of Equation 5.4 we are now able to calculate the desired temperature for a specific amount m_h. This results in the final equation

$$T_h(m_h) = \frac{18}{200}m_h + 311.15 \tag{5.6}$$



(a) Fully heated boiler. (b) Cold water pumped into boiler. (c) Boiler temperature after some time.

Figure 5.12: States of the boiler at the time of consumption.

that depends on the required amount of warm water.

5.4.3 CALCULATE__REQUIRED__TIME__FOR__HEATING

The last function in our Scheduler calculates the time needed for heating based on the desired temperature from the last function. Therefore the desired function should define the time in minutes for a specific water temperature $t(T)$. In practice water heating works by having a pipe coil inside the boiler connected to the secondary pipeline, as shown in Figure 4.1. The hot water flows through the pipe coil and emits heat to the surrounding cold water. The temperature change while heating is shown in Figure 5.13. The line of orange dots plots the true temperature values during heating when the flow temperature T_h is set at $56°C$.

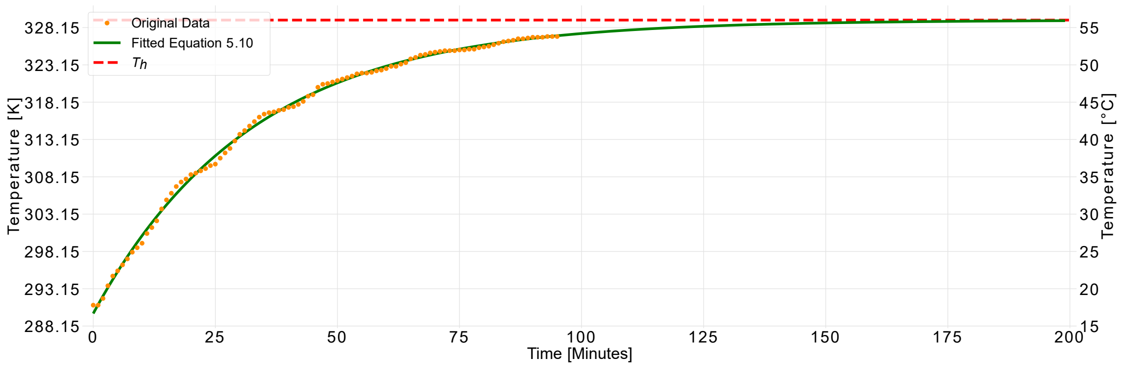


Figure 5.13: Heating curve during full heating

In order to derive a descriptive function thereof we assume that the temperature of the hot water T_h does not change as time passes. This is justified because in practice the temperature change of the hot water will be much less then the temperature change of the cold water, since the heat of the hot water gets replenished by the external heating source. Furthermore we define the temperature difference in Kelvin by

$$\Delta T = T_h - T_c. \tag{5.7}$$

In addition, we assume that the rate of temperature change is proportional to the temperature difference between the two mediums,

$$\frac{d\Delta T}{dt} = -\kappa \Delta T, \quad \kappa > 0 \tag{5.8}$$

where κ is the proportional constant which can be interpreted as an efficiency coefficient of the heat diffusion. Because of the condition

$$T_c(0) \overset{!}{=} T_c^0 \tag{5.9}$$

that states that the temperature at time 0 is the initial temperature of the cold water, Equation 5.8 has a unique solution

$$T_c(t) = T_c^0 e^{-\kappa t} + T_h(1 - e^{-\kappa t}). \tag{5.10}$$

This equation describes the temperature of the water to be heated at a specific point in time, given an initial cold temperature T_c^0, a flow temperature T_h and the constant κ.

Every necessary parameter for this equation is given except for the constant κ. So in order to find the correct value we apply a curve fitting algorithm. This algorithm uses non-linear least squares to fit the function to our measured temperature values. Since we know the values of the initial temperature and the flow temperature that produced our data we let the algorithm only adjust κ in this process. The function with the fitted κ is shown as green line in Figure 5.13. It can be noted that it almost perfectly describes the original values. The slight discrepancy can be explained by a not constant flow temperature in reality, that we assumed in our function.

The final equation that yields the time in dependence of temperature is given as

$$t(T) = \frac{1}{\kappa} \ln \left(\frac{T_h - T_c}{T_h - T} \right). \tag{5.11}$$

CHAPTER 6

Evaluation

To show the technical applicability of the serverless programming model in the edge computing paradigm we evaluate our prototypical use case implementation by defined characteristics. Specifically, the following evaluation should yield whether the serverless programming model is an appropriate computation model for the edge computing paradigm. Furthermore, it should answer the question whether a serverless edge application is more favorable for IoT scenarios than a centralized cloud implementation. To provide a comprehensive evaluation we initially present a list of characteristics that describe the serverless cloud as well as the serverless edge paradigms. We then describe our cloud reference implementation to our prototype. This reference implementation helps us to compare the serverless edge paradigm to a cloud-native IoT application. Subsequently we describe our test environment and how we gathered the data used throughout the evaluation of the individual characteristics. During this evaluation serverless edge refers to our implementation using AWS Greengrass and consequently serverless cloud refers to our cloud reference implementation. At the end we evaluate our novel consumption-oriented scheduling approach. We examine if our scheduler is able to reduce the average boiler temperature as well as the amount of heating cycles.

The claims of each characteristic, summarized from other scientific papers, are listed in Table 6.1. Besides the in-depth evaluated serverless cloud and serverless edge paradigms, Table 6.1 also lists properties of a comparable local monolithic application. It is used to highlight the benefits and/or necessities of a serverless edge application compared to long-established local monoliths deployed at each home. In the serverless cloud paradigm, a potentially very large amount of data is sent to the cloud for processing, leading to considerable bandwidth requirements [APZ18]. In contrast, the serverless edge model applies data filtering and aggregation at the edge, before data is sent to the cloud and therefore decreases the traffic to the centralized cloud [APZ18, DB16]. Responsiveness is measured by the latency on one hand and by the execution time of functions on the other hand. Ai et al. [APZ18] state that „edge computing can provide services with faster response and greater quality in comparison with cloud computing". Furthermore, because computational resources are closer to the edge, link latency is reduced to a minimum [BMNZ14, DD17]. Regarding runtime costs Baldini et al. [BCC+17] argue that the serverless programming model lowers deployment costs in the cloud because of the provided billing system. Cloud users of serverless functions are charged for execution time rather than resource allocation, as it is the case with server instances. For the same reason and further because functions are not executed in the cloud but on the edge gateway device the serverless edge paradigm is expected to incur lower runtime costs. However some one-off costs for the edge gateway device may occur. According to Baldini et al. [BCC+17] serverless programming releases the programmer of operational concerns like resource provisioning, maintenance and scalability. Moreover, they state that serverless cloud platforms „strive to make deployment as simple as possible". This corresponds to the conclusion of Bonomi et al. [BMNZ14] for edge computing platforms, as they articulate that these platforms should handle „multi-tenant distributed deployment of applications across the path between the edge and the centralized cloud" [BMNZ14]. The authors also argue that this requires inter-host migrations of applications, hence transparent support of code mobility. Because the edge is seen as an extension of the cloud, components of edge computing applications have to run both in the cloud as well as in the edge devices [DGC+16]. The research work of Garcia Lopez et al. [GME+15] focuses on a user-centric perspective of the edge computing vision. They emphasize that moving all personal and social data generated by IoT devices to centralized services implies a loss of privacy. Furthermore, they argue that the „trust is in the edge", hence sensitive data is more secure when stored in the edge.

6.1 Prerequisites

6.1.1 Cloud Reference System Implementation

Before we are able to examine the serverless edge paradigm we implement a reference system, running in the cloud. For this we migrate the existing edge computing prototype to serverless functions in the cloud as far as possible. Limitations, like the maximum execution time or maximum filesize of the unzipped function code, are circumvented by outsourcing these to web services running in virtual servers. These web services are then called from within the respective serverless function. The resulting serverless cloud architecture is illustrated in Figure 6.1. It is composed of the exact same functions as the initial serverless edge implementation, but additionally requires two web services and additional cloud services. Because a GGG encapsulates an individual home, functions executed on each GGC do not have to be aware of the home they run in. A developer can simply ignore the fact that these functions are used by different customers / homes. In contrast, the serverless cloud implementation is globally responsible for all residencies, thus each function call has to be customer-aware. Therefore we have to store our machine learning models in an AWS S3 bucket with a unique name and assign it to every residency. Furthermore, we persist per-home metadata in AWS DynamoDB — a key-value store —

Characteristic	Serverless Cloud	Serverless Edge	Local Monolithic Application
Internet Bandwidth Usage	high – all data transmitted	low – limited data transmitted	none – completely local
Responsiveness	low – dependent on latency (cloud $\leftrightarrow$ edge)	high – LAN communication	high – zero latency
Runtime Costs	high – many cloud services	low – few cloud services + inexpensive edge gateway device per home	middle – powerful device for monolith
Application Deployment	done by cloud provider	mostly done by cloud provider – initial setup required	entire setup required + manual deployment
Code Mobility	transparent execution in the cloud	transparent execution across edge and cloud	no mobility
Data Privacy	low – data in cloud	high – sensitive data stored locally	high – all data stored locally

Table 6.1: Characteristics of Serverless Cloud, Serverless Edge and Local Monolithic Application

instead of using each GGC's shadow state.

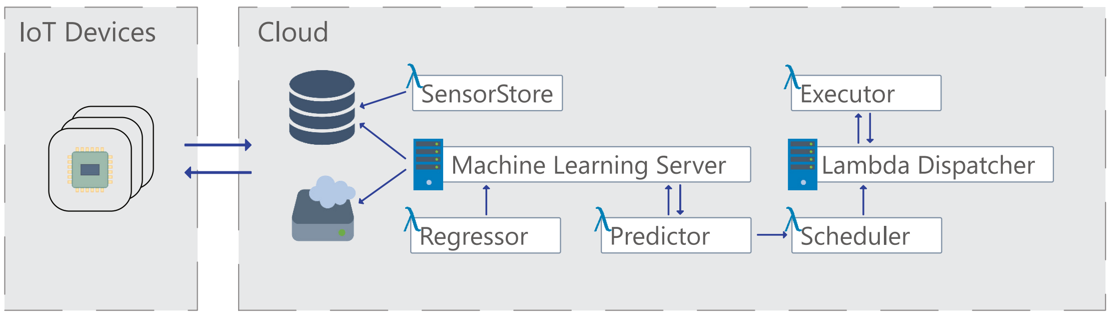


Figure 6.1: Conventional district heating

The limitations of cloud functions, as explained before, prevent us from performing machine learning directly in the Regressor and Predictor functions. First, the longer the system is in use, the more training instances are recorded and the longer is the overall time to train the model. Second, the size of the dependencies necessary for the machine learning algorithm exceeds the maximum filesize restriction. So instead we call a web service that handles the training of the model and the prediction itself. In case of the Regressor the web service trains the model with the new data asynchronously to keep execution time low, thus also reduce induced costs. The Predictor however is not affected by this, that is why it requests the web service and keeps waiting for the response. In case any water consumption is predicted it acts like the serverless edge system and invokes the Scheduler function. Subsequently if the scheduler determines to heat, the Executor function is invoked. In our comparative system the Executor function waits until scheduled time is reached to start heating and further waits until the end of the scheduled plan to stop it again. However, due to the maximum execution time limitation of cloud functions and charges per execution time, we have to separate waiting and execution phases. For this reason, and because of the lack of an existing cloud-native solution for lambda ad-hoc scheduling, we implement our own lambda dispatching service. So the Scheduler of the cloud system indirectly invokes the Executor function via the dispatching service, given the desired invocation time. On invocation the Executor then starts the heating cycle and subsequently dispatches itself to stop it again after the scheduled duration.

6.1.2 Test Environment & Data Measurement

We have deployed our serverless edge system for an entire year at our test location — a single-family home located in Austria. During this year three people lived in this home and had the instruction not to adjust their water consumption behavior due to the new water heating system. The inhabitants consisted of three adults, a student and his parents. The father has followed regular full-time work, whereas the mother has been employed for a part-time job.

During this year we gathered initial training data and improved our model to achieve the desired forecasting accuracy. Because we needed a huge data set for accurate prediction results, this lasted for about the first nine months. Subsequently we replaced the original heating system with our serverless edge implementation of the proposed consumption-oriented scheduling approach. From then onwards we started to monitor our system with respect to the listed characteristics from Table 6.1.
The methods we applied to measure all quantitative characteristics are described in the following section. In general, for both implementations, we measured all aspects for one week and calculated the averages thereof.

- *Internet Bandwidth Usage*

  We analyzed the bandwidth usage using Wireshark, a „widely-used network protocol analyzer“ [Wir17]. For gathering network communication we specifically used a command line utility, called *dumpcap*, part of Wireshark's toolkit. Once started the tool records all network communication that can be later analyzed using Wireshark. In addition to the network communication we examine the amount of transmitted IoT messages to the cloud. This can be done with AWS Cloudwatch, a centralized logging service that also gathers metrics across multiple AWS services. AWS IoT publishes metrics about sent and received messages associated with different message types. These types include the ones we are interested in: *GetThingShadow* and *UpdateThingShadow*.

- *Responsiveness*

  To measure the responsiveness of the systems we categorized the serverless functions into execution flows. These are: IoT devices' shadow updates, storing the sensor data, train the model and predict and act. We then added verbose logging to our functions. Log entries followed defined keywords like *STARTED* and *STOPPED* and a segment name, e.g., *STARTED [SensorStore]*. Based on the contained key words that marked the beginning and end of a certain segment along with precise timestamps, we were able to reconstruct the defined execution flows. A custom parser converted these log files into JSON objects representing the recursive structure of segments. Each segment contained start and end timestamps to determine the durations. In a last step we calculated the averages across all related segments. Additionally we estimated the latency between individual devices/servers with MTR [MTR] to complete execution flows distributed across multiple machines.

- *Runtime Costs*

  AWS has a cost explorer that lets you track expenses of individual services. To distinguish between costs caused by the individual systems we ran the systems independently of each other. Therefore, we used one system at a time for a whole week, while the other system was completely shutdown during that period. Afterwards we extracted the induced costs from the cost explorer grouped by used services.

	Serverless Edge			Serverless Cloud		
	WAN	LAN	Total	WAN	LAN	Total
Packets	1,317	763,264	764,581	32,014	692,609	720,169
Avg. Packets/sec	0.0152	8.8341	8.8493	0.3705	8.0163	8.3353
MB	0.1183	388.4061	388.2544	6.2151	375.1113	381.3264
Avg. Bytes/sec	1.3692	4495.441	4496.8102	71.9338	4341.5665	4413.5004
Avg. MBit/sec	0.00001	0.036	0.036	0.0006	0.0347	0.0353

Table 6.2: Average bandwidth usage per day

6.2 Characteristics

6.2.1 Internet Bandwidth Usage

The first characteristic we have a look at is the differences in bandwidth usage between a serverless cloud and a serverless edge system. Table 6.2 lists the exact measurements of packets and respective data volumes that are transmitted during a single day. These measurements are divided into transmissions between the home and cloud, referred to as WAN, and transmissions that happens between the GGC and sensors/actuators, referred to as LAN. The packet filters that were used to separate these transmissions are shown in Listing 6.1 and Listing 6.2 respectively. It is also necessary to limit the IP protocol to TCP transmission to remove all unrelated communications like ARP broadcasting or ICMP messages. The IP addresses in Listing 6.2 are the address of the devices running the GGC, the sensors and the control and monitoring interface of the heat exchanger unit.

```
!ip.src in {192.168.0.0/16 172.16.0.0/12 10.0.0.0/8 127.0.0.1} or
!ip.dst in {192.168.0.0/16 172.16.0.0/12 10.0.0.0/8 127.0.0.1} and
ip.proto == "TCP"
```

Listing 6.1: Wireshark Filter for WAN communication

```
ip.src in { 192.168.0.20 192.168.0.32 192.168.0.38 } and
ip.dst in { 192.168.0.20 192.168.0.32 192.168.0.38 } and
ip.proto == "TCP"
```

Listing 6.2: Wireshark Filter for LAN communication

As we can see the exchanged data in total are almost identical in regard to packet count and transferred data volume. The serverless edge system however, transmits only about 0.0017% of all packets to the cloud, whereas the serverless cloud sends roughly 4.5% to the cloud. The transmitted packets of the serverless edge application do not result from our prototype implementation, but rather from the AWS Greengrass framework itself. For this reason we were not able to assuredly identify the sources of the packets sent in the serverless edge application. But we assume that these packets are used to regularly update the connectivity information required for the Discovery API used in IoT devices. This means that the serverless edge application saves about 95.9% of packets sent between the cloud and each home. In terms of data volume this corresponds to a reduction of approximately 98.1%, thus only transmits 0.1183MB on average per day. This observation is also reflected in the amount of IoT messages sent. Figure 6.2 contrasts the amount of IoT activities against the cloud service of a serverless cloud application with the ones of a serverless edge application. The serverless cloud system performs in total 8,677 shadow updates and reads on average per day. In contrast, the serverless edge application only performs 12 activities per day.

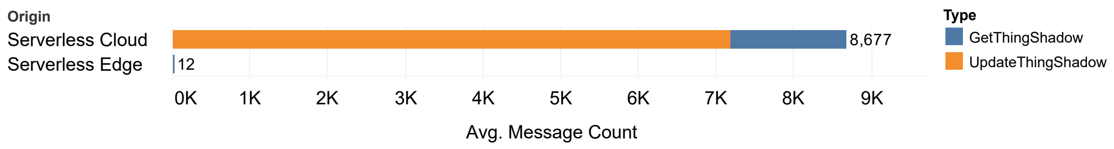


Figure 6.2: Average count of IoT-messages per day

6.2.2 Responsiveness

In Section 6.1.2 we have briefly explained our approach to soundly compare the responsiveness of both systems. For this, we split our systems into separate processes to simplify the comparison. The processes that make up the whole system are stated as follows:

- **IoT devices**

  This process includes all sensors that measure the surrounding and subsequently update their shadows. Figure 6.3 illustrates this execution flow. At first each device gathers its measurement and finally update its shadow state.

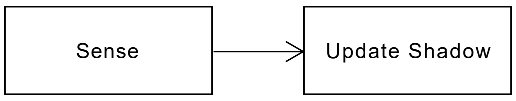


Figure 6.3: Execution flow of IoT devices

- **SensorStore**

  The SensorStore flow consists of an initial step to gather all thing-shadows. Then it persists these measurements into the database, whether it is the local database in the case of the serverless edge application or a database deployed in the cloud. This execution flow can be seen in Figure 6.4.

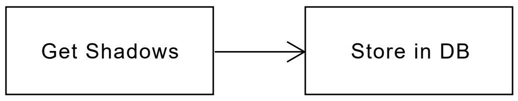


Figure 6.4: Execution flow of SensorStore function

- **Regressor**

  The Regressor flow is illustrated in Figure 6.5. It consists of initially loading all available training data. Then the function trains the model and finally stores it either on the file system or in an object storage. Latter is required for the serverless cloud implementation.

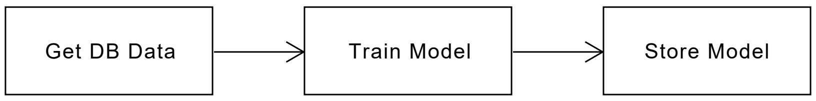


Figure 6.5: Execution flow of Regressor function

- **Predictor**

  The last execution flow involves the Predictor function and its successors — the Scheduler and Executor function. At first latest measurements are retrieved from the database and the stored model is loaded. Then the volume is predicted and consequently the Scheduler is invoked if necessary. Subsequently the Scheduler requests real-time shadow data for its scheduling algorithm. Finally the Executor function executes the plan and acts accordingly. This is shown in Figure 6.6. For the serverless cloud system this execution flow differs slightly because of the fundamental limitations mentioned in Section 6.1.1 and consequently the system's differences. The cloud Predictor function instead calls the web service that performs the initial tasks including the prediction of the volume. Furthermore, the Executor is invoked twice. Once for starting the heat cycle and once for stopping it again.

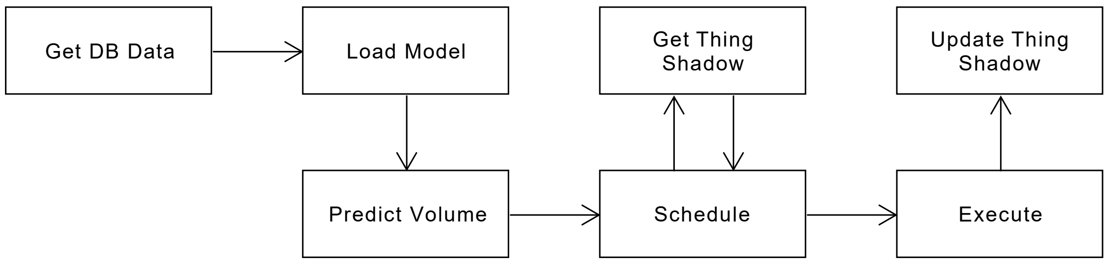
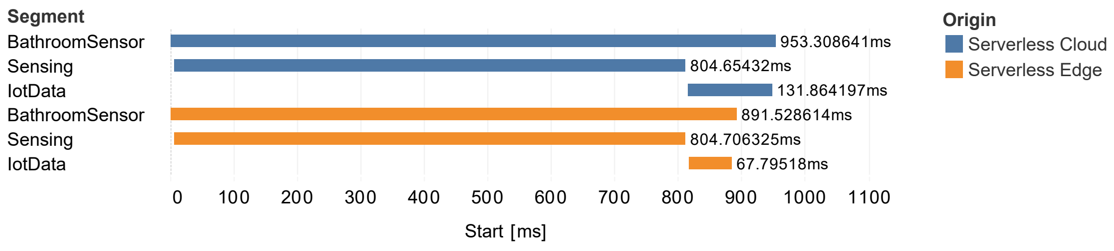


Figure 6.6: Execution flow of Predictor function including scheduling and execution

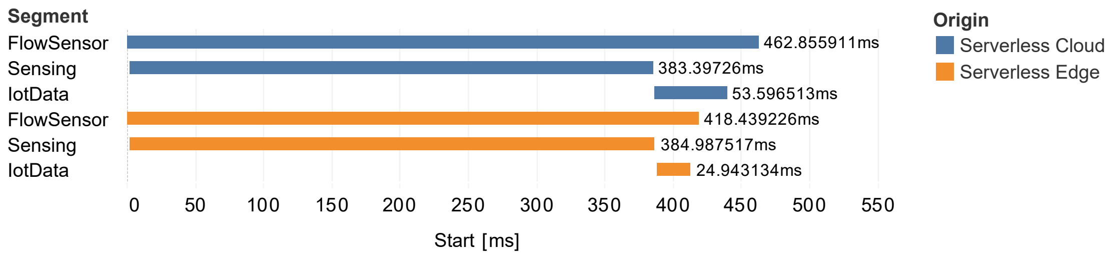
In the following we present the measurements for each of these execution flows individually. In all of the following charts the orange bars denote the segments of the serverless edge solution, whereas the blue color marks the segments of the serverless cloud system.

IoT Devices

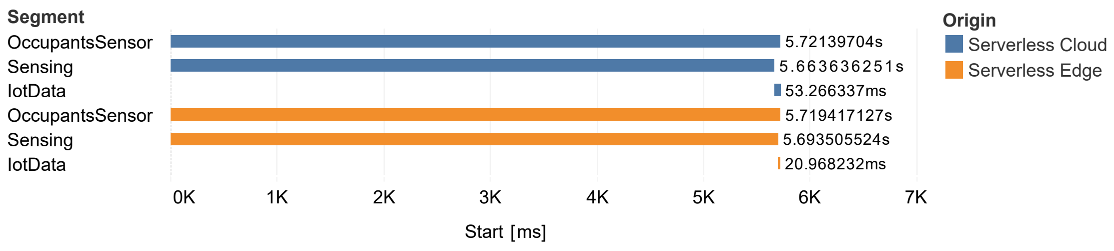
The measurements of the different IoT devices are shown in Figure 6.7. In general it can be noted that the *Sensing* segments are almost identical for all devices. These segments mark the times a device is either reading from a connected sensor or requesting data from another device via HTTP requests. Deviations, as in the Outdoor Sensor (see Figure 6.7d) and Heat Exchanger Unit (see Figure 6.7f), can be traced back to fluctuations of the response time from the control and monitoring interface. However, the most interesting segment is the *IotData* segment. It captures the duration of the shadow update MQTT request until a response arrives. It is essential to observe that in all cases the requests to the cloud take at least twice as long as the requests to the GGC in the serverless edge system. On average, shadow updates in the serverless edge variant are 55.94% faster.



(a) Bathroom Sensor

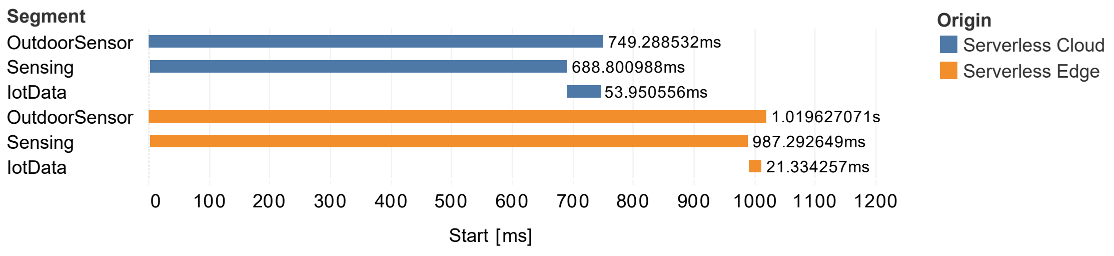


(b) Flow Sensor

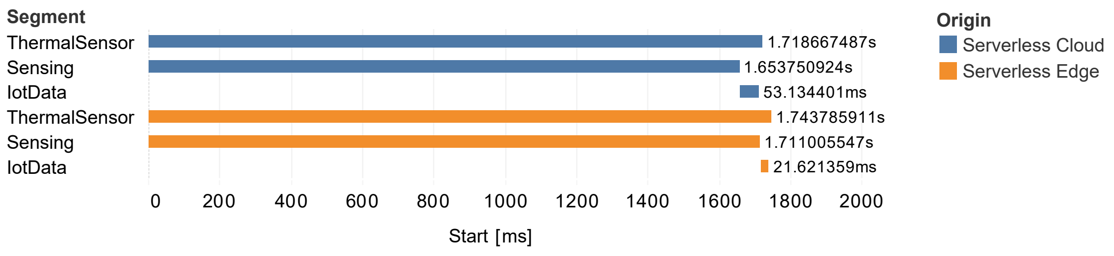


(c) Occupants Sensor

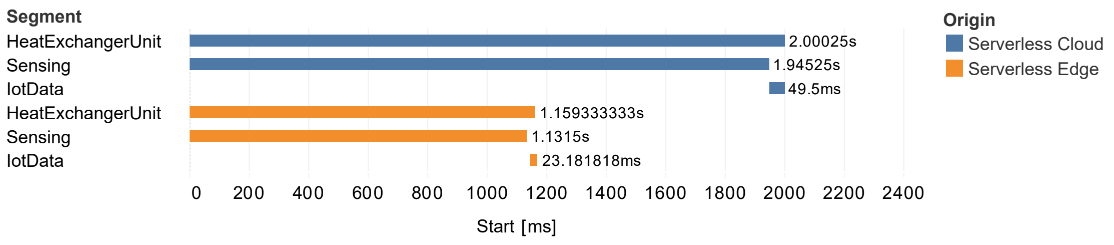
Figure 6.7: Execution Time of IoT devices



(d) Outdoor Sensor



(e) Thermal Sensor



(f) Heat Exchanger Unit

Figure 6.7: Execution Time of IoT devices (cont.)

SensorStore

As explained before, the SensorStore gathers all the shadow states from the devices and subsequently persists them into the database. So, as can be seen in Figure 6.8, the *SensorReading* segments take up about half of the execution time. The individual *IotData* segments follow the same pattern, as observed from the IoT device measurements. They take at least double the execution time in the serverless cloud implementation. However, despite intensive examination we cannot answer why the first IotData requests take much longer than the subsequent requests. The raw data did not reveal any notable statistical deviation, hence was not induced by statistical outliers.
The *SecretManager* is used to securely fetch the database user and password. But, because there was no similar functionality for AWS Greengrass at the time of writing, we fell back to a configuration file deployed with the lambda function. The SecretManager service call takes almost four times the time of the actual database interaction. Furthermore, it can also be noted that the database updates are also almost twice as fast in the edge SensorStore than in the cloud SensorStore. The reason for this is the logical distance

between the server that executes the lambda function and the database server, although both reside in the same data center. The overall runtime of the edge SensorStore is therefore roughly 82.5% on average shorter than the function running in the cloud — 150ms versus 858ms.

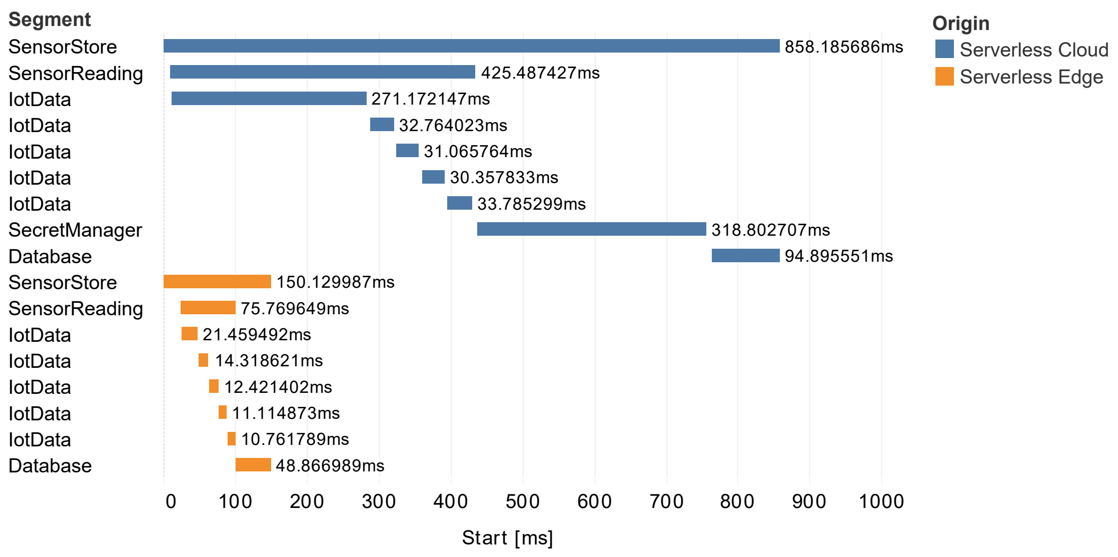


Figure 6.8: Execution Time of SensorStore

Regressor

For these average runtime calculations we set the amount of instances to 610 thousand training instances, equivalent to slightly more than one year of data. A major part of the Regressor function's runtime is used to train the model. Only about a fourth is used to load latest measurements from the database and store the model. On the GGC it takes more than 2m to fetch the data from the database, in contrast to merely 16 seconds in the cloud. Figure 6.9 also reveals a significant difference in training time. The function running at the edge gateway takes more than 8m compared to 1m25s in the cloud. As already mentioned, the cloud Regressor function does not directly perform the model training but hands off the work to a web service. This can be seen by the two Segments *MLRequest* and *MLServer*. The request to the web service does not have a significant impact on the total runtime, since the measured latency between the EC2 instance and the different servers hosting our lambda functions was only about 1ms. Storing the model into an AWS S3 bucket takes 795ms, whereas storing the model onto the local filesystem on the edge gateway takes 572ms. In the *DynamoDB* and *IotData* segments at the end of the functions we save machine learning metadata, like the mean absolute error of the model in the GGC shadow, respectively a DynamoDB table. The local shadow update again is by far faster, 13ms versus 82ms, than the update of a DynamoDB item.

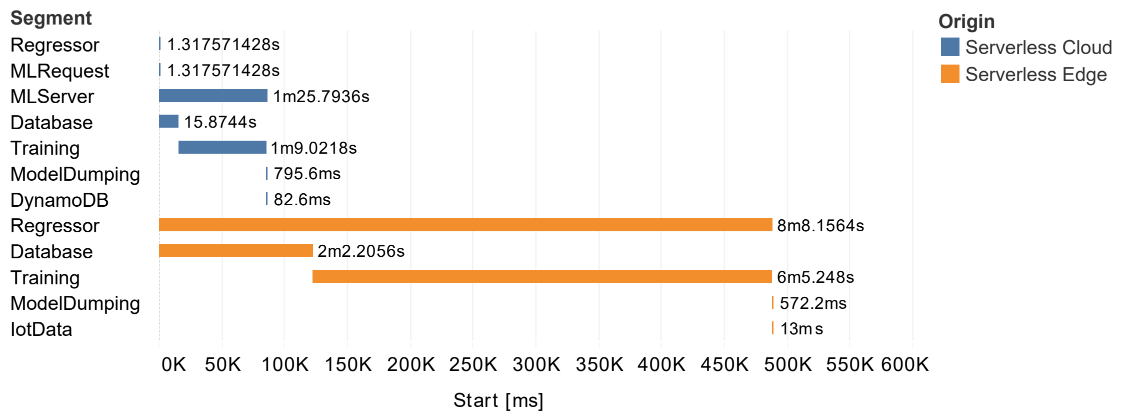


Figure 6.9: Execution Time of Regressor

Predictor

Finally we examine the runtime durations of the Predictor function. The associated measurements are shown in Figure 6.10. For accurate comparison we set the „waiting“ time of the Executor to be 1s in each case. Otherwise the Scheduler's results would influence the total runtime. The most notable things in this plot are again the different results of the machine learning tasks. The cloud Predictor waits 929ms to get a response from the web service with the prediction. The prediction itself only takes about 193ms, the rest is used for data and model loading. The same sequence in the serverless edge variant requires 6s. However, despite this huge difference, the total time until the end of the heating cycle is nearly identical. Because of the significant delay till the cloud Executor function is invoked, the total time increases to 7.9s. On the GGC this time is marginally longer with 8.4s. This delay is caused by the cold starts of the lambda function. These occur when a function is executed the first time. On subsequent calls the latency is considerably lower [HSH^{+}16], because it is then kept into memory for a finite amount of time. This time is set dynamically by AWS and is influenced by, e.g., server load and other unspecified metrics. If the function is not called within this time it has to cold start again for the next invocation. This phenomenon can also be observed when we have a look at the time from when the prediction is completed until the heat exchanger unit starts heating. We can see that the serverless edge system is a lot faster. It takes 1.2s at the GGC in contrast to 3.1s in the cloud.

6.2.3 Costs

The last examined quantitative characteristic is runtime costs. Both systems, the edge and cloud variants, required the necessary sensors / actuators that collect the data and enable the control of the water heating system. In total six sensors / actuators are necessary — a bathroom temperature sensor, a water flow sensor, an occupants sensor, an outdoor temperature sensor, a boiler temperature sensor and a device handling the

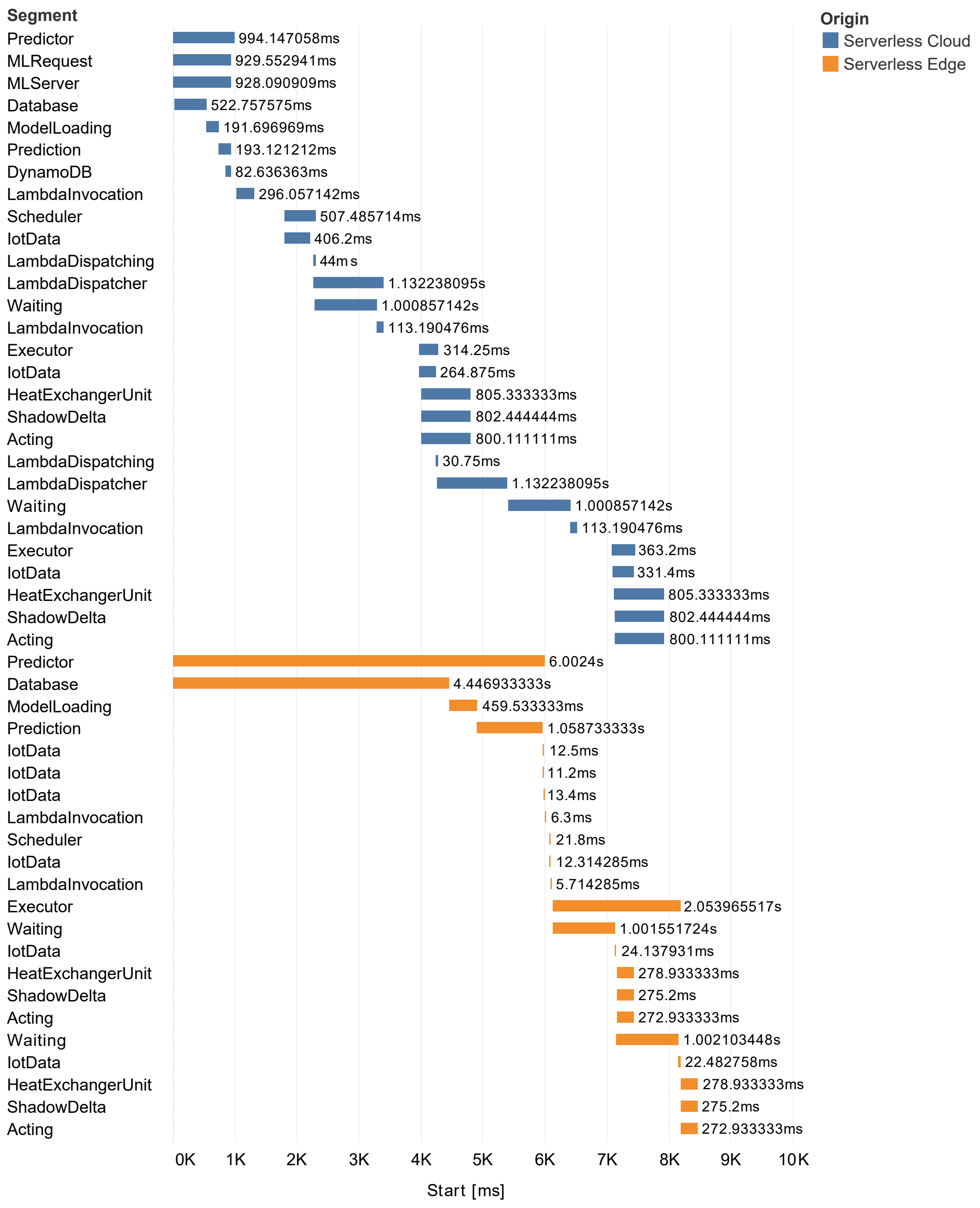


Figure 6.10: Execution Time of Predictor

heat exchanger unit. However, the serverless edge system also required a device capable of running the GGC software. For this we used a single RaspberryPi 3. Therefore the only difference in runtime costs per residency are the costs associated with the GGC device which at the time amount to roughly \$36.

A direct comparison of runtime costs for one running instance is shown in Figure 6.11. The depicted costs do not include IoT devices hardware costs, only cloud service costs. The serverless edge system only creates costs of AWS IoT and AWS S3. AWS IoT costs consist of a per-million-message price. Additionally each connected GGC device is charged per month. For large lambda functions it is recommended to upload the functions to AWS S3 beforehand. That is the reason for the extremely small expenses for AWS S3. All these costs sum up to \$0.001 per day, hence about \$0.03 per month per residency, for the serverless edge variant. The serverless cloud system in contrast uses many more AWS services. At first it requires the AWS Lambda service for our functions and a relational database, namely RDS. Further it needs at least one EC2 instance for the web services. In addition to these it requires a configured Virtual Private Cloud (VPC) with a NAT gateway that allows servers and functions in that VPC to communicate with the AWS IoT service. The VPC NAT gateway is listed as *EC2-Other*. Then it also needs a NoSQL database, in our case DynamoDB, that stores machine learning metadata for each home. Some of these are not listed in Figure 6.11 because they have an initial free-of-charge volume. All in all serverless cloud costs amount to about \$2.23 per day for a single residency, or roughly \$69 per month. With an increasing amount of residencies these costs decrease as far as these initial services can manage workload. After that, cloud services scale up accordingly.

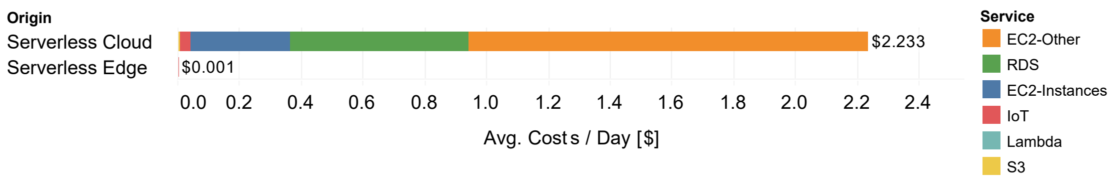


Figure 6.11: Average runtime costs per day for a single residency

6.2.4 Code Mobility

Edge computing emphasizes the universal deployment of applications across the path between the edge and the cloud. Hence applications have to be decoupled from the underlying infrastructure and support migrations between hosts. [BMNZ14, DGC+16] The serverless programming model promotes the release of operational concerns and simple deployment done by the cloud provider [BCC+17]. In addition, the high abstraction level and statelessness of serverless functions should therefore simplify inter-host migrations and automatic deployment by providers. In the following we discuss two scenarios to evaluate code mobility support in serverless edge.

Migration of an Application from Serverless Edge to Cloud

The first scenario is the migration of an application built for the serverless edge to the cloud. Our starting point is a finished implementation of our serverless edge analytics system. The system has already been deployed and successfully controls the household water heating system. The edge gateway device is malfunctioning and we want to migrate the entire system to the cloud. We describe the tasks that we had to do to make our system run in the cloud. While doing so we highlight potential obstacles and applied changes.

1. **IoT Devices**

   IoT devices in the serverless edge platform require host discovery of the GGC gateway in the local network. Therefore devices have to initially request a provided discovery API to obtain connection information about the associated group's GGC. Then they connect to the received host and start their routine workflow. IoT devices that connect to the cloud service do not require the host discovery process. Instead they pass over the discovery and directly connect to a fixed endpoint in the cloud. Despite this rather small difference a developer either has to explicitly implement fallback logic by oneself to support both variants or limit the application scope.

2. **SensorStore**

   AWS Greengrass does not support function scheduling, e.g., run a function every hour. However it supports long-living functions which can be used to implement them in a way that they execute their main tasks in desired intervals. The serverless cloud counterpart, AWS Lambda, does feature this capability with the additional Cloudwatch service. It allows developers to trigger events either at a defined rate or by providing a cron-like expression. The rest of the function is almost identical, except the missing possibility to centrally manage secrets in a secure way at the serverless edge application. For this reason we had to deploy our functions, packaged with a config file that holds our database information including username and password. However, this missing feature was then released later and is now generally available. The cloud SensorStore already leverages AWS SecretManager, a service to store and retrieve arbitrary secrets, in our case the database connection string and access details.

3. **Regressor**

   Fundamentally, the differences between the edge and cloud variants are similar to those of the SensorStore function, a long-living function versus a function regularly triggered by Cloudwatch events. A major implementation difference is induced by the lambda function's maximum execution time limitation. To circumvent this limitation we moved the machine learning part to a web service running on an AWS EC2 instance. Furthermore, because we fetch training data from the database we also leverage the SecretManager service in the cloud and fallback to a conventional but not recommended configuration file at the edge. In addition, we store customer-specific metadata of our machine learning model. For this we use a NoSQL database, specifically AWS DynamoDB at our cloud Regressor. In contrast, the edge function stores this information in the GGC shadow. Eventually the model is exported for future uses by the Predictor function. The export is done either directly on the filesystem of the GGC or by uploading to the AWS S3 object storage in the cloud.

4. **Predictor**

   The limitations of the Regressor function also apply to the Predictor function. The refactored cloud function again uses the web service to predict the water consumption. However, the rest of the code is completely identical to the edge Predictor function.

5. **Scheduler**

   The Scheduler function is not a long-living function like the SensorStore, Regressor and Predictor. Instead it is directly triggered on demand by the Predictor function. At the GGC the function performs all necessary actions — waiting, starting, waiting, stopping — by itself. However, because of the maximum function execution time that applies in the cloud the cloud function cannot run longer than 15 minutes [Ama18], hence has to be dynamically scheduled from an external service. Because there was no out-of-the-box service available we had to develop another web service for this task. This service allows us to dispatch triggers that invoke given functions at a specified time. The web service also provides the possibility to include function parameters for the function call. So on the GGC the Scheduler invokes the Executor directly whereas in the cloud the Scheduler is triggered from the web service that performs the waiting inbetween the starting and stopping phases.

6. **Executor**

   For the same reasons mentioned above the implementation changes in the Executor function focus on the waiting logic. We completely removed the waiting phases in the cloud function. Instead we let the Executor schedule itself to stop the heating again, using the dispatching web service. The rest of the code is identical to the edge function.

Migration between Serverless Edge Gateway Devices

The second scenario discusses the capability to replace an existing edge gateway device with a new one. The main difference to a cloud-migration is that we do not face difficulties due to difference in supported features, only differences due to varied infrastructure. Our initial situation in this scenario is a successfully deployed GGG on a GGC running our serverless edge analytics application. Furthermore, we have a second device running the GGC platform and a local database ready to be used. The new GGC has valid a certificate and has been set up at the AWS IoT service. Technically this means that we have a valid GGC definition, associating the new core device with the certificate and sufficient permissions. The following steps are necessary to perform the GGC migration:

1. Replace existing GGC definition with the newly created one

   This step can only be done through the command-line interface provided by AWS. To do so, we have to update the existing GGG definition and replace the old GGC definition by the new one.

2. Copy database content to the new GGC device

   In a next step we have to migrate existing data from the database to the new device. However, since the MySQL installation is entirely separate AWS Greengrass does not provide this functionality. For this reason we have to manually dump the database and import it on the new device.

3. Copy existing machine learning model to the new GGC device

   Moving the existing machine learning model to the new device also has to be done manually. However, this step can be skipped if the prediction is not required to run immediately. As soon as the Regressor retrains the model and stores it on the filesystem the Predictor resumes work.

4. Install missing libraries on new GGC device

   Besides the maximum function execution time limitation, AWS also enforces an upper bound of the unzipped function size. At the time of writing this size is 250MB. The machine learning library used in the Regressor and Predictor functions with all its dependencies is larger than this limitation. For this reason we additionally had to install the libraries directly on the GGC device and mount the Python libraries folder into our function's execution runtime. This has to be repeated on the new GGC device as well in order to successfully run both functions.

5. Redeploy GGG

   Finally, we can redeploy the GGG. This deploys the updated GGG definition to the new GGC including the entire configuration and the functions.

6. Restart IoT devices

   Lastly, to reconnect the IoT devices they need to fetch the connectivity information of the new GGC device. This can be achieved by simply restarting the sensors and actuators. On restart each device fetches the new information from the Discovery API and subsequently connects to the newly assigned gateway.

6.2.5 Data Privacy

The final characteristic in our evaluation on the serverless programming model at the edge is *Data Privacy*. We examine the privacy protection from a user-centric perspective. For this we compare the different data storage mechanism in the serverless edge as well as in the serverless cloud. We investigate where certain information is stored and draw conclusions from this knowledge. Furthermore, we discuss possible limitations in functionality because of data privacy design decisions.

Table 6.3 summarizes the various systems used for the individual information. The serverless cloud system uses multiple storage solutions in the cloud. Global anonymous statistics across homes and sensor measurements are stored separately in two relational databases in the cloud. Additionally an object storage is used for the trained models of the homes. For the persistence of machine learning metadata that we calculate on each regression we use AWS DynamoDB, a NoSQL database. Finally AWS IoT stores the latest version of each device's shadow state. However, it is not public which kind of storage mechanism is used for that purpose. Nonetheless we can state that any information collected from the system is distributed in the cloud. A user has to trust the company that sells these systems to comply with data privacy laws and to ensure data safekeeping.
In contrast, the serverless edge is able to keep data closer to the user. In our serverless edge application the entire generated sensor data is stored locally on the GGC device. Continuous sensor data is persisted in a local MySQL database. Anonymized data for global statistics of water consumption or other big data analytics is stored in the cloud using AWS RDS. Instead of the AWS S3 object storage the machine learning model is stored on the GGC filesystem. Additionally, the related metadata is saved in the GGC shadow state that in turn is also persisted on the GGC with other IoT devices' shadow documents in a SQLite database by default. So any non-anonymous information is kept in the local network of the home and is not transmitted to the cloud. However, users still depend on the company. Users do not have a simple way to verify that no data is transmitted or to stop undesired transmission.

6.3 Use case Evaluation

Finally we evaluate our use case implementation, specifically our novel consumption-oriented scheduling approach. For this we first examine the boiler temperature before and after enabling our scheduler. However, the PVS can slightly skew results because water can be heated despite not using the district heating system. Nonetheless a lower average temperature means that less energy has been used for heating. A second indicator for reduced energy consumption is the amount of total heating cycles or the total time of heating in a period of time. We investigate heating time of two months using the old system and two months using the new scheduling. Subsequently we compare the average heating time per day.

	Serverless Cloud	Serverless Edge
Global Statistics	AWS RDS	AWS RDS
Sensor Data	AWS RDS	GGC MySQL
Machine Learning Model	AWS S3	GGC Filesystem
Machine Learning Metadata	AWS DynamoDB	GGC Shadow State
IoT Device Shadow States	AWS IoT	GGC SQLite

Table 6.3: Storage mechanisms in Serverless Cloud and Serverless Edge

6.3.1 Boiler Temperature

Figure 6.12 shows the average boiler temperature throughout a day. The red line marks the temperature with the static heating approach of the old system. Compared to the blue line which represents the consumption-oriented scheduling approach it is almost $8°C$ higher at any time. Table 6.4 confirms the positive impact of the new approach. The overall average boiler temperature is — with $48.2°C$ — $8.2°C$ lower than before.
According to the calculated standard deviation the previous system normally operated at a boiler temperature of 50 to $62.9°C$, whereas the new approach usually keeps the boiler temperature between 41.6 and $54.8°C$.
Some further interesting aspects are the 75% quantiles and the high difference in the maximum temperature. In 75% of the time the temperature is lower than $61.6°C$ for static heating. In contrast, our consumption-oriented scheduling keeps the temperature below $51.3°C$ in 75% of the time. The maximum temperature lists the biggest statistical outliers. Normally the boiler temperature is capped at $55°C$ at the heat exchanger. The only reason for such high temperatures is the PVS. We can conclude that either fewer sunny days have occurred or the boiler temperature is significantly lower when the PVS starts feeding energy into the boiler. Nevertheless, a lower maximum temperature spares boiler materials and hence increases boiler lifetime.

6.3.2 Heating Time

Finally we discuss the total amount of heating cycles and a potential save of heating time. For this Table 6.5 summarizes statistics of heating behavior while using each system for two months, specifically 59 days. The table shows a total count of 130 heating cycles compared to 97 heatings with our novel scheduler which means a decrease of 25.4% (or 2.2 heatings per day, in contrast to 1.6 heatings per day). Mean heating duration is nearly identical — 18m45s compared to 16m08s. However, especially the standard

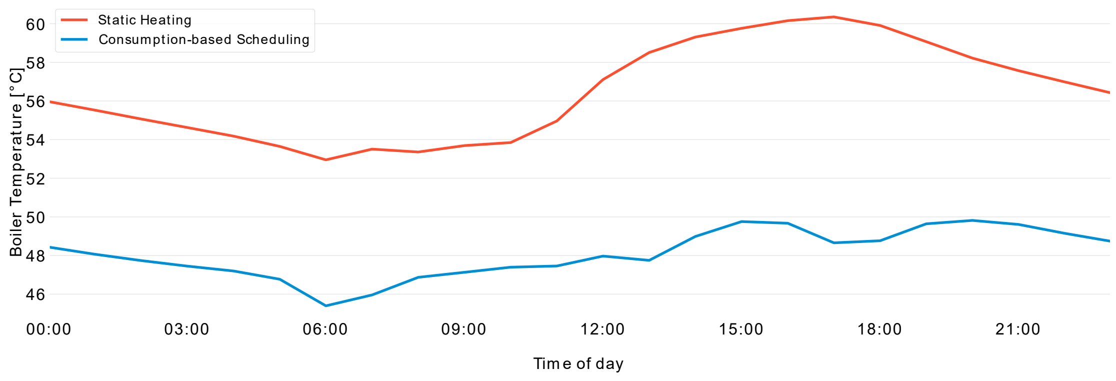


Figure 6.12: Boiler temperature before and after switching to consumption-oriented scheduling

	Static Heating	Consumption-oriented Scheduling
Mean	56.44397	48.19434
Std	6.438236	6.608566
Min	17.8	16.1
25%	52.3	44.6
50%	56.2	47.9
75%	61.6	51.3
Max	81.9	69.0

Table 6.4: Boiler temperature statistics before and after switching to consumption-oriented scheduling in $°C$

deviation is higher with the old system. We can derive that the traditional system had considerably more short-term heatings. This is also reflected in the analysis of quantiles. The minimum and 50% quantile values are identical, but the 25% quantile in the static heating measured 6m compared to 11m. In total the static heating heated for as long as 1d 16:38:01 within two months. In the same timespan the consumption-oriented scheduling has executed heatings that lasted 1d 02:44:03. Hence we could reduce heating time by about 34.2%. Converted to an average time per day this means about 41m with the static heating system and 27m with the consumption-oriented scheduling. This

	Static Heating	Consumption-oriented Scheduling
Count	130	97
Mean	0 days 00:18:45.238461	0 days 00:16:08.6907221
Std	0 days 00:16:17.152458	0 days 00:09:19.104163
Min	0 days 00:01:00	0 days 00:01:00
25%	0 days 00:06:00	0 days 00:11:00
50%	0 days 00:13:00	0 days 00:13:00
75%	0 days 00:30:00	0 days 00:18:00
Max	0 days 01:34:01	0 days 00:47:00
Sum	1d 16:38:01	1d 02:44:03

Table 6.5: Heating time statistics before and after switching to consumption-oriented scheduling. Data represents two months of each system.

reduction in heating time can be also observed in Figure 6.13. The figure shows the cumulative heating duration with respect to the system runtime of 59 days. In the first 14 days both systems behaved almost identical, however after that the consumption-oriented scheduling approach constantly heated less than the previous static heating system.

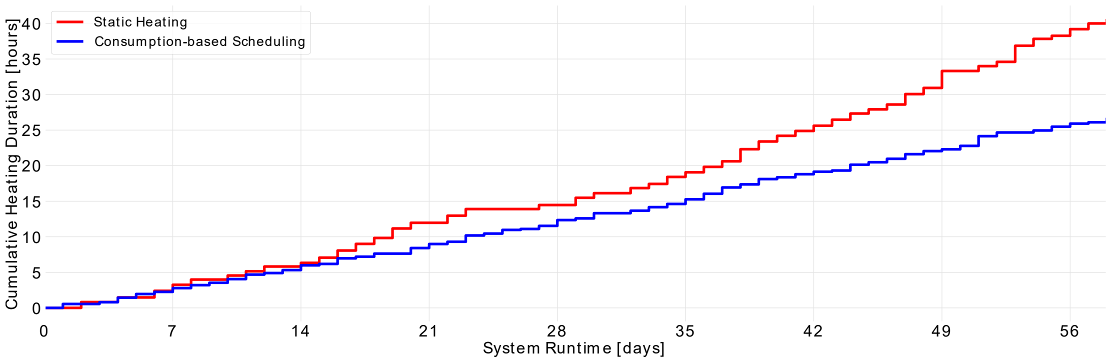


Figure 6.13: Heating time before and after switching to consumption-oriented scheduling

CHAPTER 7
Discussion

The results presented in Chapter 6 demonstrate the benefits and shortcomings of the serverless edge computing paradigm. In this Chapter, we discuss these results and consequently proof the claims summarized in Table 6.1.

The tabular evaluation, presented in Table 6.2, evidently show that the internet bandwidth usage has been significantly reduced. Only 1.4 Bytes/sec are sent to the cloud in the serverless edge computing model. In contrast, the serverless cloud paradigm required 71.9 Bytes/sec. This bandwidth usage is per residency, hence 1000 homes would produce 6.2 GB of traffic each day using a serverless cloud system. Counted up this is a traffic of 2.2TB per year. The serverless edge solution would only produce 43.2GB of traffic per year. These numbers are evidence that the claim, a considerably lower internet bandwidth usage (see Table 6.1), is true. A serverless edge system transmits a lot less data than the serverless cloud counterpart.

Regarding the responsiveness we observe that IoT message transfers are generally faster in the serverless edge system. Moreover, function invocations, as can be seen in the Predictor execution flow, take much less time. Another proof of these observations is the SensorStore function. Its execution is 82.5% faster at the edge than in the cloud. The Predictor execution flow is also much faster, when we only consider the subsequent lambda invocations and shadow updates after the CPU-intensive prediction. The machine learning tasks are constantly faster in the cloud variant by an approximate factor of 5 at the Regressor and the Predictor function. We argue that serverless edge partially fulfills the claim from Table 6.1 with respect to responsiveness in the case of message transmission and reactivity. However, CPU-bound tasks are much slower, hence diminish the edge computing benefit of low-latency. Clearly, however, this will depend on the type of edge computer used. With that said, the generalized argument of low-latency at the edge is only significant in scenarios that are not bound to the processor. A cloud server can scale vertically to increase CPU power, whereas an edge gateway's compute power is limited and generally cannot be scaled on demand.

The virtually unlimited, on-demand scalability of the cloud comes at a cost. Figure 6.11 presents the cost structures of the serverless edge and serverless cloud systems. It has to be noted that cloud services like EC2 instances, RDS, DynamoDB and specifically the NAT gateway can be used for many residencies, thus become cheaper with the amount of enrolled homes. The evaluation of runtime costs yields additional costs of \$2.23 per day for the initial cloud services. However, these expenses do not include services with a free-of-charge volume. For these reasons the serverless cloud has high stepped fixed costs, hence is relatively expensive with less residencies. With more homes using the cloud system, these extra charges decrease but the free-of-charge volumes are depleted at the same time. Therefore the discount on a per-home basis also decrease, which in turn raises costs again. For higher quantities a detailed cost structure analysis is required. [Eiv17] examines the economics of serverless cloud computing and presents a real-world case study that yields the same conclusions. In contrast, the serverless edge has lower variable costs. These can be split into low one-off costs for the GGC device and negligible monthly costs of \$0.03. Therefore, we conclude that the runtime costs claim from Table 6.1 is proven true.

The considerable difference in costs is not only caused by the need for more cloud services, but also due to necessary changes in the implementation. To evaluate the code mobility we kept track of barriers we noticed while moving our serverless edge system to the serverless cloud. Almost all implementation differences can be traced back to certain predefined limitations of cloud services from the provider's side or missing features on one side that have to be replaced for the other. This lack in features or one-sided limitations hinder code mobility. The main obstacles we observed during the migration is presented in the following:

1. Long-living functions vs. Maximum execution time
2. User-centric view vs. Global view
3. GGC Filesystem vs. AWS S3

The first and most important barrier in code mobility is the possibility of long-living functions on the GGC. Highly beneficial at the edge, but not directly supported in the cloud. For this reason it requires major code and sometimes also architectural changes. The second biggest obstacle is the difference in function scope. At the GGC a function only handles an individual user, thus can be developed without heavy logic for multi-user support. However, the cloud functions need to distinguish between users on each invocation to associate water consumption predictions and execute the schedule for the correct user. The third obstacle is state management of functions. You cannot use AWS S3 at the edge without fallbacks if you have to support full-functionality during connection losses. A fallback, like the local GGC filesystem, is not available in the cloud, hence also requires additional modification. These three obstacles have to be removed for full support of code mobility and transparent migration between edge devices and cloud servers. In addition to the migration between the two domains — edge and cloud — we also evaluated the migration of an entire GGG between two GGC devices. With regard to code mobility we can state that the serverless functions can be migrated to a new edge gateway without any implementation changes. Nonetheless, some steps have to be carried out manually, including database and model migration as well as manual installation of necessary dependencies for functions that exceed the maximum lambda size limitation. When we look at both scenarios as a whole we can state that code mobility with serverless edge is only supported to a small extent. Functions can be executed on different servers or devices in the respective domain, but cannot be migrated from one domain to another without major manual intervention. We argue that the claim, the support for transparent execution of functions across the edge and the cloud, is not fulfilled in today's production-grade serverless edge frameworks.

In addition to code mobility we also evaluated the *Data Privacy* in serverless edge applications. For this we examined the various storage mechanisms used in both implementation types. These are listed in Table 6.3. It can be noted that most of the data in the serverless edge application is stored locally on the GGC. Additionally, anonymized and, e.g., aggregated data can still be used for big data analytics in the cloud without decreasing data privacy. However users of IoT services, no matter if implemented within the serverless edge or serverless cloud paradigms, still have to trust the companies to keep their data safe. There is no simple way for non-expert users to keep track of what data is processed or even persisted in the cloud. Moreover, users do not have control over their data. In summary, we conclude that the serverless edge improves data privacy by reducing the amount of private data that is stored in the cloud, but it is still a matter of trust and data privacy is not directly enforceable by the users.

All our results on the claims from Table 6.1 are summarized in Table 7.1.

7.1 Limitations

7.1.1 Serverless Edge

Three of our defined characteristics are only partially fulfilled using the serverless edge paradigm. The responsiveness is only improved with respect to latency, but as soon as the edge gateway executes CPU-bound tasks the overall execution takes considerably more time than in the cloud. So the effectiveness of processing data closer to the source is decreased. This impact can be seen in Section 6.2.2, in particular at the Regressor execution flow.
Another significant aspect of the serverless edge is code mobility of functions. It enables the execution of functions on different nodes. The qualitative evaluation however revealed problems when migrating edge functions to the cloud. In the edge framework's current state functions are too coupled to specific functionalities of the GGC. This prevents direct migration to the cloud and results in the need for two implementations of the function — one for the edge and one for the cloud. Directly related to this is the present deployment mechanism. Functions are either deployed to a GGC or not. The cloud service does not

Characteristic	Serverless Edge
Internet Bandwidth Usage	yes
Responsiveness	partial
Runtime Costs	yes
Application Deployment	partial
Code Mobility	no
Data Privacy	partial

Table 7.1: Evaluation results

automatically scale or migrate functions inbetween a GGC and the cloud.
In addition, we noticed limitations in data privacy handling from a user's perspective. Overall, the results show that the serverless edge provides the capabilities to enhance data privacy. It significantly reduces or even completely prevents sensitive data from being sent to the cloud without affecting service functionality. However, the serverless framework does not provide a control mechanism to enforce the data privacy settings of a user. Users have to trust the companies to keep their data safe.

7.1.2 Serverless Framework

Besides the unmet characteristics we also faced several technical limitations. The framework itself does not come with an integrated storage system besides the IoT device shadows. If edge applications require additional persistence systems like relational databases or similar, they have to be installed explicitly on the edge gateways. Additionally, AWS lambda functions have a filesize limitation of the unzipped function code. At the time of writing it is 250MB. But because the libraries for data aggregation, data preparation and machine learning in sum are bigger than this limit we were not able to deploy the Regressor and Predictor functions to our GGC. To work around this limitation we had to install the libraries on the GGC device beforehand and include them at runtime. All these issues further hinder code mobility and migration.

Another major obstacle during development was software testing. Splitting application code into small atomic functions should considerably increase testability. AWS provides testing and debugging for lambda functions through AWS Serverless Application Model (SAM). It enables local testing and step-through debugging of serverless applications. However, edge functions depend on several GGC features and as already stated do not run in the cloud without modification. This renders edge functions untestable locally. The only way to test correct behavior, is to deploy the functions to a GGC and conduct manual testing. Moreover, debugging is not available and someone can only use systematic logging output to find errors.

On the IoT devices side there is the need to call a *Discovery API* to retrieve GGC connectivity information. However, this imposes problems if there is no internet connection at the moment when sensors / actuators try to connect to the GGC. Even if the local network is intact, sensors and actuators are still not able to connect to each other. So, if an application is required to work flawlessly on internet outages IoT devices have to explicitly cache gathered connectivity information. Besides this, a developer is also responsible for the implementation of ordinary connection retry logic. These rather important features are not included in IoT devices SDK provided by AWS.

To sum up, AWS Greengrass provides serverless functions and analytics at the edge, but nonetheless misses some important aspects of edge computing and serverless programming. The serverless functions deployed at the edge gateway is too coupled to the GGC and does not abstract away all infrastructural aspects. Furthermore, edge computing facilitates multi-tenancy of edge devices, hence enable deployment of multiple vertical applications from different cloud users. At the moment, AWS Greengrass, as well as the other evaluated serverless edge frameworks, are not built to host multiple strictly separated applications on their edge gateways. This is especially not the case if these applications belong to distinct cloud users. Therefore every cloud user has to deploy an edge gateway for its own application, hence we claim that these frameworks are only for bespoke applications.

7.2 Consumption-oriented Scheduling

Finally we discuss the results from the use case evaluation in Section 6.3. The results clearly show a considerable improvement, both in average boiler temperature and total heating time. Moreover, despite a lower boiler temperature, on average by $8.2°C$, inhabitants did not complain about water temperature for personal hygiene. However sometimes the water temperature was slightly too low for dish-washing, that in general requires hotter water than showering. This points out that the desired boiler temperature is not only dependent on the required amount of water, but also on the temperature that is required for a specific task. To further improve the scheduler, it must be possible to differentiate between, e.g., showering and dish-washing. This would allow for more precise temperature calculations. A possible solution for this would be to add water temperature sensors at the sinks and build another machine learning model that predicts required water temperature for a specific task or enables profiling of water consumption. The result from this could be integrated in the formula for boiler temperature calculations.

The total heating time was also drastically reduced by a third (see Table 6.5 and Figure 6.13). However, the heating consists of two phases and our scheduler is only capable to calculate the time of the last phase and uses a constant time for the other.

The first phase is to heat up the primary pipeline (see Figure 4.1) from the district heating plant to the heat exchanger unit. As soon as the external circulation system has reached sufficient temperature the secondary pump is activated. Not before this, the water in the boiler is heated. For this reason, the heating time is dependent on the time that is required to finish the first phase and after that to actually heat the boiler. The scheduler's calculation of time only considers the second phase and approximates the first phase by a constant time. In reality this time varies greatly. The primary pipeline is almost continuously warm in winter, due to active radiators and underfloor heating. This significantly reduces the required time in winter. In summer, the complete opposite is the case. So to further improve the scheduler it would be necessary to know the water temperature of the primary pipeline and estimate a heating duration thereof.

Both discussed limitations refer to the accuracy of the boiler temperature and heating time calculations in the scheduler. Another limitation is the accuracy of the forecasting model itself. Due to the irregular nature of water consumption an exact prediction is extremely complex. Furthermore, abnormal high water consumption causes peaks and downgrade the machine learning model. At the same time, the model lacks the possibility to accurately predict these peaks. But nonetheless, the current results show that the combination of PVSs, conventional heating and a smart consumption-oriented scheduling can save energy and reduce costs.

CHAPTER 8

Conclusion

The IoT is constantly growing and expected to exceed 24 billion devices in 2020 [GBMP13]. The cloud computing paradigm provides virtually unlimited processing power and storage, hence is a key element of today's IoT to support this vast amount of devices [LL15]. However, the Cloud of Things [AKAH14] faces limited data transfer rate, requires continuous Internet connection and causes transmission of a significant data volume. The edge computing paradigm brings data processing and storage closer to the user, hence enables latency sensitive applications, reduces bandwidth and fosters scalability and reliability. The heterogeneous and resource-constrained runtime environments of the edge of the network require light-weight application encapsulation to support code mobility and inter-host migration. For this reason [NRS+17, dLGL+16, MBS+17] propose the serverless programming model as suitable programming and infrastructure abstraction model.

However, besides proposed serverless edge analytics architectures, there have only been minimal real-world use case evaluations that examine the serverless edge computing paradigm. Therefore we evaluated the serverless edge model using a production-grade edge computing framework. We compared the serverless edge to the serverless cloud paradigm by quantitative and qualitative metrics defined in Table 6.1. Prior to this we defined an IoT use case and built two respective prototypes. The use case consists of general IoT application building blocks — environment monitoring, home appliance control to fulfill set goals, data analytics on historic data as well as real-time data analytics. Our prototypes implement a novel consumption-oriented scheduling approach for water heating. The systems leverage district heating and a PVS and optimizes heating cycles by domestic water consumption profiling on historic data. For this we built a random forest model and combine it with real-time information to determine necessity of heating and further schedule next heating.

In a next step we compared available serverless edge computing frameworks, specifically AWS Greengrass and Microsoft Azure Iot Edge. Subsequently we described a generic serverless edge analytics architecture and applied it to our use case implementation using the selected framework. We then presented our random forest machine learning model, the independent variables thereof and the deployment in our test installation. Then, we defined an acceptance criteria and validated our model using time series cross-validation. Concluding our prototype design we illustrated the novel consumption-oriented scheduling approach. Therein we presented the derivation of required mathematical formulas and the utilization in our scheduler.

To evaluate both paradigms we deployed the serverless edge system in a single-family home. Before we enabled our scheduler, we initially collected data to train our model and achieve defined accuracy. At the same time we migrated our serverless edge system to the cloud and subsequently replaced the serverless edge system in our test environment. We then compared both implementations by defined characteristics (see Table 6.1). Specifically internet bandwidth usage, responsiveness, runtime costs, application deployment, code mobility and data privacy. The results showed that the serverless edge has the potential to significantly reduce bandwidth requirements and runtime costs. The evaluation also confirmed the presumed decrease in latency. However, CPU-bound tasks diminish the improvement because we found that these tasks can be computed considerably faster in the cloud than on resource-constrained edge gateways. We also identified issues that hinders code mobility to support automatic application deployment and scaling across the edge and the cloud. Moreover, we observed that at the time of writing production-grade frameworks do not support multi-tenancy, hence only enable bespoke edge applications. Finally, the qualitative evaluation revealed that it is not possible to seemlessly distribute applications across edge and cloud without additional development efforts.

We argue that the serverless programming model is a suitable edge computation model. However, production-grade frameworks lack some important features, identified in our evaluation, to fully realize the idea of edge computing. Foremost is the support of a multi-tenant architecture on edge gateways to mitigate limitations to specifically customized applications and provide better general access for cloud users. For this, research on multi-tenant edge gateway architectures has to be done. Additionally, current frameworks do not entirely abstract away infrastructure details, which prevents code mobility and transparent execution on cloud or edge. To facilitate seemless migration between hosts we argue that research on some type of service discovery has to be performed. The service discovery endpoint should then provide mechanisms to switch for example between cloud storage and local file storage in a transparent way. However, in order to enable transparent service replacement, edge frameworks additionally need to be able to perform data migration, such that application states can be replicated on different hosts. This includes replication of local databases and local files. Research in these three fields would remove edge application development obstacles and foster global accessibility to edge computing for cloud users. With regard to the identified data privacy concerns, scientific work could develop methods to include IoT devices / application users into the control loop to oversee and perhaps restrict transmission of sensitive data. Restrictions could then be a decisive factor for serverless function deployment.

List of Figures

List of Tables

List of Algorithms

Acronyms

RFID Radio-Frequency IDentification. 8

RSSI received signal strength indicator. 19

SaaS Software-as-a-Service. 7, 13

SAM Serverless Application Model. 72

VPC Virtual Private Cloud. 60

WSN Wireless Sensor Network. 8

Bibliography

[AAA$^+$17] Hany F. Atlam, Ahmed Alenezi, Abdulrahman Alharthi, Robert J. Walters, and Gary B. Wills. Integration of Cloud Computing with Internet of Things: Challenges and Open Issues. In *2017 IEEE International Conference on Internet of Things (iThings) and IEEE Green Computing and Communications (GreenCom) and IEEE Cyber, Physical and Social Computing (CPSCom) and IEEE Smart Data (SmartData)*, pages 670–675. IEEE, jun 2017.

[AAGES10] Nesreen K Ahmed, Amir F Atiya, Neamat El Gayar, and Hisham El-Shishiny. An Empirical Comparison of Machine Learning Models for Time Series Forecasting. *Econometric Reviews*, 29(56):594–621, 2010.

[AIM10] Luigi Atzori, Antonio Iera, and Giacomo Morabito. The Internet of Things: A survey. *Computer Networks*, 54(15):2787–2805, oct 2010.

[AKAH14] Mohammad Aazam, Imran Khan, Aymen Abdullah Alsaffar, and Eui Nam Huh. Cloud of Things: Integrating Internet of Things and cloud computing and the issues involved. In *Proceedings of 2014 11th International Bhurban Conference on Applied Sciences and Technology, IBCAST 2014*, pages 414–419. IEEE, jan 2014.

[Ald03] Frances K. Aldrich. Smart Homes: Past, Present and Future. In *Inside the Smart Home*, pages 17–39. Springer-Verlag, London, 2003.

[Amaa] Amazon Web Services. AWS Greengrass.

[Amab] Amazon Web Services. AWS IoT.

[Amac] Amazon Web Services. AWS Lambda – Serverless Compute.

[Ama18] Amazon Web Services. AWS Lambda enables functions that can run up to 15 minutes, 2018.

[AN06] Jc Augusto and Cd Nugent. Smart homes can be smarter. *Designing Smart Homes: the Role of Artificial Intelligence*, pages 1–15, 2006.

[APK16] Kaiser Ahmed, Petri Pylsy, and Jarek Kurnitski. Hourly consumption profiles of domestic hot water for different occupant groups in dwellings. *Solar Energy*, 137:516–530, nov 2016.

[APZ18] Yuan Ai, Mugen Peng, and Kecheng Zhang. Edge computing technologies for Internet of Things: a primer. *Digital Communications and Networks*, 4(2):77–86, apr 2018.

[AWW18] Hany Atlam, Robert Walters, and Gary Wills. Fog Computing and the Internet of Things: A Review. *Big Data and Cognitive Computing*, 2(2), apr 2018.

[BBB$^+$13] Gianluca Bontempi, Souhaib Ben Taieb, Yann-Aël Borgne, Souhaib Ben Taieb, and Le Borgne. Machine Learning Strategies for Time Series Forecasting. In *Business Intelligence*, volume 138, pages 62–77. 2013.

[BCC$^+$17] Ioana Baldini, Paul Castro, Kerry Chang, Perry Cheng, Stephen Fink, Vatche Ishakian, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Aleksander Slominski, and Philippe Suter. Serverless computing: Current trends and open problems. In *Research Advances in Cloud Computing*, pages 1–20. Springer Singapore, Singapore, 2017.

[BCF$^+$17] Ioana Baldini, Perry Cheng, Stephen J. Fink, Nick Mitchell, Vinod Muthusamy, Rodric Rabbah, Philippe Suter, and Olivier Tardieu. The serverless trilemma: function composition for serverless computing. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software - Onward! 2017*, pages 89–103, New York, New York, USA, 2017. ACM Press.

[BMNZ14] Flavio Bonomi, Rodolfo Milito, Preethi Natarajan, and Jiang Zhu. Fog computing: A platform for internet of things and analytics. *Studies in Computational Intelligence*, 546:169–186, 2014.

[BMZA12] Flavio Bonomi, Rodolfo Milito, Jiang Zhu, and Sateesh Addepalli. Fog computing and its role in the internet of things. In *Proceedings of the first edition of the MCC workshop on Mobile cloud computing - MCC '12*, page 13, New York, New York, USA, 2012. ACM Press.

[Bre01] Leo Breiman. Random forests. *Machine Learning*, 45(1):5–32, 2001.

[Bun17] Bundesministerium für Wissenschaft Forschung und Wirtschaft. Energie in Österreich 2017 - Zahlen, Daten, Fakten. Technical report, Bundesministerium für Wissenschaft, Forschung und Wirtschaft, 2017.

[CIMS17] Paul Castro, Vatche Ishakian, Vinod Muthusamy, and Aleksander Slominski. Serverless Programming (Function as a Service). In *Proceedings - International Conference on Distributed Computing Systems*, pages 2658–2659. IEEE, jun 2017.

[CKY08] Rich Caruana, Nikos Karampatziakis, and Ainur Yessenalina. An empirical evaluation of supervised learning in high dimensions. In *Proceedings of the 25th international conference on Machine learning - ICML '08*, pages 96–103, New York, New York, USA, 2008. ACM Press.

[Cri] CRIU.

[CZ16] Mung Chiang and Tao Zhang. Fog and IoT: An Overview of Research Opportunities, dec 2016.

[DB16] Amir Vahid Dastjerdi and Rajkumar Buyya. Fog Computing: Helping the Internet of Things Realize Its Potential. *Computer*, 49(8):112–116, aug 2016.

[DD17] Koustabh Dolui and Soumya Kanti Datta. Comparison of edge computing implementations: Fog computing, cloudlet and mobile edge computing. In *GIoTS 2017 - Global Internet of Things Summit, Proceedings*, pages 1–6. IEEE, jun 2017.

[DGC$^+$16] Amir Vahid Dastjerdi, H. Gupta, R. N. Calheiros, S. K. Ghosh, and Rajkumar Buyya. Fog Computing: Principles, architectures, and applications. In *Internet of Things: Principles and Paradigms*, pages 61–75. Elsevier, 2016.

[Die02] Tg Dietterich. Machine learning for sequential data: A review. *Structural, syntactic, and statistical pattern recognition*, pages 1–15, 2002.

[dLGL$^+$16] Eyal de Lara, Carolina S. Gomes, Steve Langridge, S. Hossein Mortazavi, and Meysam Roodi. Poster abstract: Hierarchical serverless computing for the mobile edge. In *Proceedings - 1st IEEE/ACM Symposium on Edge Computing, SEC 2016*, pages 109–110. IEEE, oct 2016.

[Doc] Docker - Build, Ship, and Run Any App, Anywhere.

[Dos17] Christopher Dossman. Top 6 errors novice machine learning engineers make, 2017.

[Eiv17] Adam Eivy. Be Wary of the Economics of "Serverless" Cloud Computing. *IEEE Cloud Computing*, 4(2):6–12, mar 2017.

[EJ12] Johan Eker and Jorn W. Janneck. Dataflow programming in CAL - Balancing expressiveness, analyzability, and implementability. In *Conference Record - Asilomar Conference on Signals, Systems and Computers*, pages 1120–1124. IEEE, nov 2012.

[Eva11] Dave Evans. The Internet of Things - How the Next Evolution of the Internet is Changing Everything. *CISCO white paper*, (April):1–11, 2011.

[FAS18] E. Fuentes, L. Arce, and J. Salom. A review of domestic hot water consumption profiles for application in systems and buildings energy performance analysis, jan 2018.

[GBMP13] Jayavardhana Gubbi, Rajkumar Buyya, Slaven Marusic, and Marimuthu Palaniswami. Internet of Things (IoT): A vision, architectural elements, and future directions. *Future Generation Computer Systems*, 29(7):1645–1660, sep 2013.

[GME$^+$15] Pedro Garcia Lopez, Alberto Montresor, Dick Epema, Anwitaman Datta, Teruo Higashino, Adriana Iamnitchi, Marinho Barcellos, Pascal Felber, and Etienne Riviere. Edge-centric Computing. *ACM SIGCOMM Computer Communication Review*, 45(5):37–42, sep 2015.

[HSH$^+$16] Scott Hendrickson, Stephen Sturdevant, Tyler Harter, Venkateshwaran Venkataramani, Andrea C Arpaci-Dusseau, and Remzi H Arpaci-Dusseau. Serverless Computation with openLambda. In *Proceedings of the 8th USENIX Conference on Hot Topics in Cloud Computing*, HotCloud'16, pages 33–39, Berkeley, CA, USA, 2016. USENIX Association.

[IBM] IBM Cloud. Watson IoT Platform Edge.

[JSMP04] Nanyan Jiang, Cristina Schmidt, Vincent Matossian, and Manish Parashar. Enabling Applications in Sensor-based Pervasive Environments. In *Proc. of BaseNets in conjunction with Broadband Communications, Networks, and System (BroadNets)*, volume 21, pages 871—-883, 2004.

[JSP06] Nanyan Jiang, Cristina Schmidt, and Manish Parashar. A decentralized content-based aggregation service for pervasive environments. In *Proceedings for ICPS:2006 International Conference on Pervasive Services*, volume 2006, pages 203–212. IEEE, 2006.

[Kal17] David Kaleko. From Neutrinos to Data Science, 2017.

[LL15] In Lee and Kyoochun Lee. The Internet of Things (IoT): Applications, investments, and challenges for enterprises. *Business Horizons*, 58(4):431–440, jul 2015.

[Lxc] LXC / Linux Containers.

[MB17] Garrett McGrath and Paul R. Brenner. Serverless Computing: Design, Implementation, and Performance. In *Proceedings - IEEE 37th International Conference on Distributed Computing Systems Workshops, ICDCSW 2017*, pages 405–410. IEEE, jun 2017.

[MBS$^+$17] Amardeep Mehta, Rami Baddour, Fredrik Svensson, Harald Gustafsson, and Erik Elmroth. Calvin Constrained -A Framework for IoT Applications in Heterogeneous Environments. *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, pages 1063–1073, jun 2017.

[Mica] Microsoft Azure. Azure Functions - Serverless Architecture.

[Micb] Microsoft Azure. Azure IoT Edge.

[Micc] Microsoft Azure. Stream Analytics - Real-time data analytics.

[Moz05] Michael C. Mozer. Lessons from an Adaptive Home. In *Smart Environments: Technology, Protocols and Applications*, pages 271–294. John Wiley & Sons, Inc., Hoboken, NJ, USA, jan 2005.

[MTR] MTR.

[NPM+16] Yannis Nikoloudakis, Spyridon Panagiotakis, Evangelos Markakis, Evangelos Pallis, George Mastorakis, Constantinos X. Mavromoustakis, and Ciprian Dobre. A Fog-Based Emergency System for Smart Enhanced Living Environments. *IEEE Cloud Computing*, 3(6):54–62, nov 2016.

[NRS+17] Stefan Nastic, Thomas Rausch, Ognjen Scekic, Schahram (TU Wien) Dustdar, Marjan Gusev, Bojana Koteska, Magdalena Kostoska, Boro (Ss. Cyril Jakimovski, Methodius University), Sasko Ristov, and Radu (University of Innsbruck) Prodan. A Serverless Real-Time Data Analytics Platform for Edge Computing. 2017.

[PVG+12] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, Jake Vanderplas, Alexandre Passos, David Cournapeau, Matthieu Brucher, Matthieu Perrot, and Édouard Duchesnay. Scikit-learn: Machine Learning in Python. *Journal of Machine Learning Research*, 12:2825–2830, 2012.

[RAD] Thomas Rausch, Cosmin Avasalcai, and Schahram Dustdar. Portable Energy-Aware Cluster-Based Edge Computers. Technical report.

[RDMP17] Eduard Gibert Renart, Javier Diaz-Montes, and Manish Parashar. Data-Driven Stream Processing at the Edge. In *Proceedings - 2017 IEEE 1st International Conference on Fog and Edge Computing, ICFEC 2017*, pages 31–40. IEEE, may 2017.

[RGN+18] Amir M. Rahmani, Tuan Nguyen Gia, Behailu Negash, Arman Anzanpour, Iman Azimi, Mingzhe Jiang, and Pasi Liljeberg. Exploiting smart e-Health gateways at the edge of healthcare Internet-of-Things: A fog computing approach. *Future Generation Computer Systems*, 78:641–658, jan 2018.

[Ric76] Georg Wilhelm Richmann. Novi Commentarii academiae scientiarum Petropolitanae. 20:189–207, 1776.

[RMD$^+$06] Vincent Ricquebourg, David Menga, David Durand, Bruno Marhic, Laurent Delahoche, and Christophe Logé. The smart home concept: Our immediate future. In *2006 1st IEEE International Conference on E-Learning in Industrial Electronics, ICELIE*, pages 23–28. IEEE, dec 2006.

[Sat17] Mahadev Satyanarayanan. The emergence of edge computing. *Computer*, 50(1):30–39, jan 2017.

[SBCD09] Mahadev Satyanarayanan, Paramvir Bahl, Ramón Cáceres, and Nigel Davies. The case for VM-based cloudlets in mobile computing. *IEEE Pervasive Computing*, 8(4):14–23, oct 2009.

[SCZ$^+$16] Weisong Shi, Jie Cao, Quan Zhang, Youhuizi Li, and Lanyu Xu. Edge Computing: Vision and Challenges. *IEEE Internet of Things Journal*, 3(5):637–646, oct 2016.

[TW10] Lu Tan and Neng Wang. Future Internet: The Internet of Things. In *ICACTE 2010 - 2010 3rd International Conference on Advanced Computer Theory and Engineering, Proceedings*, volume 5, pages V5–376–V5–380. IEEE, aug 2010.

[VRM14] Luis M Vaquero and Luis Rodero-Merino. Finding your Way in the Fog: Towards a Comprehensive Definition of Fog Computing. *ACM SIGCOMM Computer Communication Review*, 44(5):27–32, 2014.

[VS17] Prateeksha Varshney and Yogesh Simmhan. Demystifying Fog Computing: Characterizing Architectures, Applications and Abstractions. In *Proceedings - 2017 IEEE 1st International Conference on Fog and Edge Computing, ICFEC 2017*, pages 115–124. IEEE, may 2017.

[WGB99] M. Weiser, R. Gold, and J. S. Brown. The origins of ubiquitous computing research at PARC in the late 1980s. *IBM Systems Journal*, 38(4):693–696, 1999.

[Wir17] Wireshark. Wireshark · Go Deep., 2017.

[WRS$^+$17] Shaoen Wu, Jacob B. Rendall, Matthew J. Smith, Shangyu Zhu, Junhong Xu, Honggang Wang, Qing Yang, and Pinle Qin. Survey on Prediction Algorithms in Smart Homes. *IEEE Internet of Things Journal*, 4(3):636–644, jun 2017.

[WSJ15] Roy Want, Bill N. Schilit, and Scott Jenson. Enabling the internet of things. *Computer*, 48(1):28–35, jan 2015.

[WSLW15] Wei Wu, Wenxing Shi, Xianting Li, and Baolong Wang. Air source absorption heat pump in district heating: Applicability analysis and improvement options. *Energy Conversion and Management*, 96:197–207, may 2015.

[XHF+17] Xiaomin Xu, Sheng Huang, Lance Feagan, Yaoliang Chen, Yunjie Qiu, and Yu Wang. EAaaS: Edge Analytics as a Service. In *Proceedings - 2017 IEEE 24th International Conference on Web Services, ICWS 2017*, pages 349–356. IEEE, jun 2017.

[YHQL15] Shanhe Yi, Zijiang Hao, Zhengrui Qin, and Qun Li. Fog Computing: Platform and Applications. *2015 Third IEEE Workshop on Hot Topics in Web Systems and Technologies (HotWeb)*, pages 73–78, nov 2015.

[YLL15] Shanhe Yi, Cheng Li, and Qun Li. A Survey of Fog Computing: Concepts, Applications and Issues. In *Proceedings of the 2015 Workshop on Mobile Big Data - Mobidata '15*, pages 37–42, New York, New York, USA, 2015. ACM Press.

[YMSG+14] M. Yannuzzi, R. Milito, R. Serral-Gracia, D. Montero, and M. Nemirovsky. Key ingredients in an IoT recipe: Fog Computing, Cloud computing, and more Fog Computing. In *2014 IEEE 19th International Workshop on Computer Aided Modeling and Design of Communication Links and Networks (CAMAD)*, pages 325–329. IEEE, dec 2014.