TU WIEN Informatics

Serverless Edge Computing für industrielle Prozesse

DIPLOMARBEIT

zur Erlangung des akademischen Grades

Diplom-Ingenieur

im Rahmen des Studiums

Software Engineering & Internet Computing

eingereicht von

David Scherer, BSc.
Matrikelnummer 11777743

an der Fakultät für Informatik

der Technischen Universität Wien

Betreuung: Univ.Ass. Pantelis Frangoudis, PhD
Mitwirkung: Manuel Hofbauer, DI. BSc.

Wien, 27. Jänner 2025

David Scherer	Pantelis Frangoudis

TU WIEN Informatics

Serverless Edge Computing for Industrial Processes

DIPLOMA THESIS

submitted in partial fulfillment of the requirements for the degree of

Diplom-Ingenieur

in

Software Engineering & Internet Computing

by

David Scherer, BSc.
Registration Number 11777743

to the Faculty of Informatics

at the TU Wien

Advisor: Univ.Ass. Pantelis Frangoudis, PhD
Assistance: Manuel Hofbauer, DI. BSc.

Vienna, January 27, 2025

David Scherer	Pantelis Frangoudis

Erklärung zur Verfassung der Arbeit

David Scherer, BSc.

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten und Abbildungen –, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.
Ich erkläre weiters, dass ich mich generativer KI-Tools lediglich als Hilfsmittel bedient habe und in der vorliegenden Arbeit mein gestalterischer Einfluss überwiegt. Im Anhang „Übersicht verwendeter Hilfsmittel“ habe ich alle generativen KI-Tools gelistet, die verwendet wurden, und angegeben, wo und wie sie verwendet wurden. Für Textpassagen, die ohne substantielle Änderungen übernommen wurden, haben ich jeweils die von mir formulierten Eingaben (Prompts) und die verwendete IT- Anwendung mit ihrem Produktnamen und Versionsnummer/Datum angegeben.

Wien, 27. Jänner 2025

David Scherer

Danksagung

Zuallererst möchte ich Pantelis Frangoudis für seine außergewöhnliche Unterstützung und Betreuung danken, insbesondere in der letzten Phase dieser Arbeit. Er hatte immer ein offenes Ohr für meine Anliegen, versorgte mich mit vielen nützlichen Informationen und nahm sich viel Zeit für mich. Es war eine wahre Freude, mit ihm zu arbeiten.

Ich möchte mich auch bei Dr. Carina Schlögl, Manuel Hofbauer und Michael Denk dafür bedanken, dass sie mir die Möglichkeit gegeben haben, gemeinsam an einem sehr interessanten Problem am LKR zu forschen. Sie gaben mir alles, was ich für die Forschung brauchte und unterstützten mich gut.

Acknowledgements

First of all, I want to thank Pantelis Frangoudis for his exceptional support and supervising, especially during the last phase of this thesis. He always paid a lot of attention to my concerns, provided me with lots of interesting ideas and useful feedback, and took his time for me. It was a real pleasure working with him.

I also want to thank the people from AIT, namely Dr. Carina Schlögl, Manuel Hofbauer and Michael Denk, for giving me the possibility to research together on a very interesting problem at LKR. They gave me everything i needed to conduct this research and supported me well.

Kurzfassung

Industrie 4.0 zielt darauf ab, industrielle Prozesse durch die Einführung intelligenter Edge-Systeme zu verbessern, die durch Datenerfassung und -verarbeitung von Sensorwerten unterstützt werden. Dies verbessert die Effizienz, Kommunikation und Flexibilität in industriellen Prozessen. Insbesondere die Echtzeitverarbeitung von Sensordaten kommt den Fabriken besonders zugute, da sie den Einsatz datenbasierter Optimierungstechniken ermöglicht, wie z. B. den Einsatz von Modellen des maschinellen Lernens zur Erkennung von Anomalien in der Fertigung. Serverless Computing hat das Potenzial, eine wichtige Rolle als Grundlagentechnologie in der industriellen Automatisierung zu spielen, da es den flexiblen Einsatz von serviceorientierten Software-Architekturen (die möglicherweise auch die Anpassung von Produktlinien erleichtern) mit feiner Granularität ermöglich, sowie eine effiziente Ressourcenskalierung. Vor allem in ressourcenbeschränkten Edge-Szenarien kann Serverless ein entscheidender Faktor sein. Im Falle plötzlicher und unvorhersehbarer Spikes in der der Rechenlast hilft die Möglichkeit, verschiedene Anbieter zu nutzen, die Belastung des Gesamtsystems zu reduzieren und einen reibungslosen Betrieb zu gewährleisten. Zur Maximierung der Leistung von Serverless im Bereich der Industrie zu maximieren, nutzen wir eine hybride Edge-Cloud Herangehensweise und schlagen einen intelligenten Serverless-Workload-Scheduler vor, der auf Deep Reinforcement Learning (DRL) basiert. Dieser wählt auf Grundlage spezifischer Anforderungen die am besten geeignetste Serverless-Instanz. Dieser Scheduler ist ein integraler Bestandteil unserer End-to-End-Systemarchitektur, die wir entwerfen und implementieren, mit dem Ziel, Industrie 4.0-Anwendungsszenarien zu unterstützen. Unsere Architektur konzentriert sich auf effiziente Datenaufnahme und Interoperabilität und unterstützt den weit verbreiteten OPC UA-Standard für die Erfassung von Sensordaten aus Industrieanlagen. Aufbauend auf dem NGSI-LD-Standard, erleichtert dies die Integration von Applikationen, indem der Datenaustausch standardisiert wird. Wir demonstrieren die Machbarkeit und industrielle Relevanz unseres Ansatzes, indem wir ihn auf einem realistischen Anwendungsfall aus dem Bereich der anwenden. Außerdem zeigen wir, dass unser DRL-basierter Scheduler die Gesamtleistung der serverless Funktionsbereitstellung Abhängigkeit von definierten Prioritäten wie Zuverlässigkeit, Reaktionszeit, Abschwächung von Cold-Starts, und/oder Vermeidung von Überlastungen der Prozessoren.

Abstract

Industry 4.0 aims to enhance industrial processes by introducing smart edge systems, leveraged by data collection and processing of sensor readings. This improves efficiency, communication and flexibility in industrial processes. Especially, real-time processing of sensor data benefits factories as it enables the use of data-based optimization techniques, such as deploying machine learning models for detecting anomalies in manufacturing. Serverless computing has the potential to play an important role as an enabling technology in industrial automation, as it can facilitate the flexible deployment of service-oriented software architectures (potentially also facilitating customization of product lines) with a fine granularity and the ability to scale resources appropriately following compute demand that may vary over time. Especially in resource-limited edge scenarios, serverless can act as a crucial enabler. In case of sudden and unpredictable spikes in computation load, having the option to use different providers helps to reduce computing stress on the overall system and ensures frictionless operation. In order to maximize performance of serverless in industrial scenarios, we leverage a hybrid edge-cloud setting and propose an intelligent serverless workload scheduler based on Deep Reinforcement Learning (DRL), which decides on the best-fitting serverless instance based on specific requirements. This scheduler is an integral component of an end-to-end system architecture that we design and implement, with the aim of supporting Industry 4.0 application scenarios. Our architecture focuses on efficient data ingestion and interoperability, supporting the widely adopted OPC UA standard for sensor data collection from industrial equipment and building on the NGSI-LD standard to facilitate application integration and data exchange. We demonstrate the feasibility and industrial relevance of our approach by applying it to a realistic use case from the domain of lightweight metal manufacturing. Furthermore, we show that our DRL-based scheduler improves overall serverless function serving performance significantly depending on the defined priorities, such as function invocation reliability, response time, cold start mitigation and/or avoiding CPU overload.

Contents

CHAPTER 1

Introduction

1.1 Motivation

The concept of Industry 4.0 poses a fundamental transformation in the manufacturing landscape. This fourth industrial revolution integrates cyber-physical systems, the Internet of Things (IoT), edge and cloud computing, and cognitive technologies to create intelligent, interconnected factories and aid process automation and end-to-end digital value chain creation. This enables more control and less errors in the systems. Especially, remote locations like offshore oil factories benefit from such interconnection. Therefore, industries aim to move from human-controlled processes to autonomous processing [HS24]. Industry 4.0 promises significant advancements in industrial processes, including the following:

- **Enhanced Efficiency and Productivity:** Real-time data analytics causing a better production planning and better resource allocation results in an efficiency gain in manufacturing processes. Furthermore, real-time data processing and evaluation can improve product quality and save resources used in manufacturing processes, thus reducing overall costs.
- **Increased Communication and Collaboration:** Industry 4.0 also establishes solid and predictable processes where resulting data can be used for precise planning and accurate estimates which leads to a high stakeholder satisfaction. Different processes also exchange data between each other which enables automatic decision making.
- **Flexibility and Customization:** Due to a highly connected and intelligent infrastructure, systems are able to adapt to specific needs which enables customization of product lines.

Due to the distributed nature of Industry 4.0, edge computing plays an important role here. Compared to cloud computing, processing data on the edge provides several benefits such as real-time data processing enabled through its low latency characteristics and enhanced privacy, as data is not sent to third party cloud providers, but is kept locally. However, using the cloud may still be an option to consider because of its scalable processing power, availability and fault tolerance, thus leading to the idea of a hybrid approach. In case of the edge server being overloaded, the cloud serves as a viable and reliable alternative due to it's ability of having "infinite" processing power. By having this as a fallback option, it is ensured that the system will be able to continue running during manufacturing processes in case of high load.

Another benefit is that the cloud could be used to offload insensitive tasks there, i.e. tasks which do not have security or time constraints, thus saving local computing power for critical tasks. An example use case, which we will address in this thesis, is real time prediction of machine sensor data, where sensors produce large amounts of data which need to be processed and provided to an anomaly detection machine learning model (ML). Therefore, having the cloud as an offloading option, more machines would be able to send data to the edge server and get their data evaluated.

Factories of the future, within Industry 4.0 and towards Industry 5.0 [LSW+22], are characterized by a "softwarization" of manufacturing processes as also evidenced by the emergence of the concepts of Digital Twins and Digital Shadows [KBBK22, WW24, BDJ+22]. Such complex software cannot be monolithic, but rather should follow a microservices-based design, and eventually be cloud-native in the sense that it can be deployed and executed over general purpose compute infrastructure, which can be provided on demand by public or private clouds.

Serverless computing can have an important role as an enabling technology here, as it can facilitate the flexible deployment of such service-oriented software architectures (potentially also facilitating customization of product lines) with a fine granularity (function level vs. micro-service/component level) and the ability to scale resources appropriately following compute demand that may vary with time. Especially in resource-limited edge scenarios, serverless can act as a crucial enabler. In case of sudden and unpredictable spikes in computation load, having the option to use different providers helps to reduce computing stress on the overall system and ensures frictionless operation [HS24]. In comparison to a microservice architecture, serverless computing can be a better fit in specific cases, due to its function-level granularity, which enables better control over which tasks to schedule outside the main system. Serverless, often in the form of Function as a Service (FaaS) and combined with Backend-as-a-Service (BaaS) offerings, stands for a technique which does not involve developers maintaining servers by themselves. They rather focus on functionality by writing code only, where an underling system manages these function calls. Typical benefits include better and simpler scaling, cost efficiency due to its pay-per-use nature (similar efficiency in an on-premise setting), simplified development and management, high availability and its lightweight property.

However, Industry 4.0 is still a developing and emerging concept that has not yet been fully

explored. Furthermore, various companies are striving for digitalization and optimization of their processes in order to use different kinds of resources more efficiently, thus allowing them to keep up with the competition. Adopting techniques of Industry 4.0 into existing processes is challenging and raises critical questions about system architecture, cybersecurity, monitoring and maintainability aspects of such a distributed system.

Specifically, the concrete problem of this thesis is connecting different parts of manufacturing processes together into a smart end-to-end system, leveraging on the capabilities of Industrial IoT (IIoT), state-of-the-art interoperability standards, and serverless computing. By simply forwarding sensor values, other applications would struggle using that kind of data as classifying sensor values, gathered from various machines and sensors within a factory, without additional context, is hard. This interoperability challenge needs to be handled in a standardized way. Therefore, standards such as NGSI-LD [Pri21] might provide a remedy, enforcing adding information about data. We refer to this as *context*. For instance, machine sensors (IoT devices) produce a large volume of data which could be used for real-time processing and automatic optimization. This could be realized by creating such a system within a serverless edge setting, where functions (FaaS) are utilized to handle individual data processing tasks per machine or IoT-sensor. This thesis will address specific architectural and algorithmic challenges, as described in the following section.

This thesis explores a new approach to serverless function scheduling, developed in collaboration with the Austrian Institute of Technology (AIT) and its subcompany, Leichtmetallkompetenzzentrum Ranshofen (LKR). LKR provided a real-world use case in light metal manufacturing, integrating an anomaly detection model into a serverless architecture for their casting process. This model, deployed as a serverless function, identifies manufacturing deviations, enabling real-time, automated adjustments based on sensor data, including autonomous process shutdowns if needed. This integration, driven by LKR's Industry 4.0 initiative, optimizes resource use, enhances product quality, and demonstrates the practical impact of this research.

1.2 Aim of the Work

In this thesis we will adopt techniques and concepts of Industry 4.0 to support manufacturing processes. While the research questions we pose have a more general applicability in smart manufacturing scenarios, our work will be driven by, tailored to, and evaluated on a real-world industrial use case provided by LKR. The architecture that we design and implement can be applied to support and transform LKR's legacy manufacturing system and will feature concepts and approaches from serverless edge computing.

In particular, the aim of this work is enabling the use of data-based optimization techniques for various manufacturing processes in a serverless on-premise edge setting. In current legacy systems, such as the one involved in our use case, machines already produce sensor data which are saved and sent to the system for evaluation afterwards, i.e. *offline*. The problem here is that such manufacturing processes take several hours

to complete, and use of course a significant amount of resources (time, energy and light metals) which might lead to a waste of resources and time if anything goes wrong during the process. Additionally, for research purposes, a company such as the LKR running experiments is likely to encounter inaccuracies, e.g., in initial configurations, where real-time data processing and monitoring become crucial. A real-time processing approach could possibly save resources in case of a production inaccuracy caused by a machine, by adjusting or completely stopping the process, thus saving energy and materials. This thesis sets off to answer the following research questions:

RQ1: How can a serverless approach be applied in an industrial IoT edge setting, what benefits can it bring about, and how can existing standards for data collection and representation be leveraged to improve on integration capabilities and interoperability?

We aim to explore this specific serverless architecture design in the context of Industrial IoT. We believe that it might be highly beneficial to be able to adapt processing functions, thus allowing to extend functionality easily, e.g. using different data-driven algorithms towards extending the cognitive capabilities of the system. Due to this, this approach might also be viable for other domains besides light metal manufacturing. Importantly, this thesis addresses several interoperability challenges, as, for example, manifested in the need to ingest in the system data originating at diverse machinery and when specific industrial standards are put in place for data acquisition, such as OPC-UA.[1] In addition, the thesis explores the adoption of modern IoT data interoperability standards, particularly the ETSI NGSI-LD specification[2] and the FIWARE framework.[3]

RQ2: How to achieve high-performance function serving in Industrial IoT in order to meet requirements for real-time data processing?

Here our aim is to explore techniques for high-performance serverless function serving particularly to meet low latency, high availability, and overall system efficiency requirements. A promising approach is to combine edge and cloud resources to schedule serverless function workloads dynamically. Particularly, we apply Deep Reinforcement Learning (DRL) techniques to effectively balance between edge and cloud execution based on application and system requirements, and the current system state (e.g. cloud offloading in case of high edge load, to save costs or for computationally heavy tasks). Answering RQ2 also involves looking into the *cold start* problem in FaaS, which can be more severe in edge settings and where solutions to alleviate it are highly relevant.

[1] https://opcfoundation.org/about/opc-technologies/opc-ua/

[2] https://www.etsi.org/deliver/etsi_gr/CIM/001_099/008/01.02.01_60/gr_CIM008v010201p.pdf

[3] https://www.fiware.org/

RQ3: How does such a system affect performance in IIoT systems where performance is a critical key consideration?

Our last question consists in evaluating if and to what extent such a complex solution improves the overall system performance, whether this system is suitable for IIoT scenarios with real-time requirements, and if our approach is feasible and industrially relevant. Answering this question involves experimental work in two directions: (i) Measuring the benefits and overheads specific to our workload scheduling algorithms, and (ii) evaluating our system design and implementation end-to-end on a realistic industrial use case, aiming to also investigate if there are hidden bottlenecks introduced by our system components and, if so, how they can be mitigated.

1.3 Contributions

This thesis presents two primary contributions to the field of industrial automation in the context of Industry 4.0.

1.3.1 Architecture

Firstly, we propose a novel, end-to-end microservice architecture designed to facilitate the seamless integration of industrial processes with Industry 4.0 principles. This architecture prioritizes extensibility and maintainability, leveraging the inherent advantages of microservice design. Moreover, it incorporates serverless technologies to further enhance scalability and flexibility.

At a high level, our system architecture is as follows: Data from diverse factory machinery is aggregated at a gateway and transmitted to a context broker, which uses the NGSI-LD standard [BBFL19] to provide crucial context about the data to applications that wish to make use of it. The adoption of the NGSI-LD standard enriches the data with comprehensive contextual information, thereby simplifying data utilization for client applications. The context broker then stores data persistently. Since we target industrial IoT settings, the de facto communication protocol is OPC UA, which many modern machine sensors of various domains implement. Client applications can access this data either through direct subscription to the context broker or by querying. In our target IIoT scenario, serverless functions are not bound to a specific deployment location. Rather, we address *multi-provider* serverless settings, where functions can be hosted at the serverless platforms of different providers (public clouds and on-premise FaaS platforms included). We come up with the idea of a more sophisticated scheduling mechanism, thus introducing a function scheduler component in our design. This scheduler operates at a higher level, outside the FaaS platform, and can be integrated with client applications or be deployed as a single stand-alone component. We discuss the FaaS scheduling problem in more detail in the background chapter followed by thorough evaluation.

We demonstrate the feasibility of our approach by implementing the proposed system and applying it to an industrial IoT use case in the area of light metal manufacturing.

We then evaluate the practicality of the end-to-end system in a local deployment with simulated sensor values. To ensure real world applicapility, we measure sensor patterns and values from real machine sensors in light metal casting.

1.3.2 Serverless Function Execution

Secondly, we introduce a serverless function scheduler that employs the Deep Q-Network (DQN) algorithm, a DRL mechanism, to dynamically allocate incoming function invocations to the most suitable serverless provider from a predefined set. This dynamic scheduling capability allows for the mitigation of cold starts. Furthermore, it enables the optimization of function execution based on pre-configured parameters, such as latency, function labels, or user-defined execution preferences. Thus, it offers a higher degree of control and customization over function execution compared to other methods, which primarily rely on server-side predictions based on metrics like function instance count and CPU utilization to mitigate cold starts only, as outlined in [ARB21, FR24].

We carry out extensive simulations with our DRL-based scheduling mechanism and show it to outperform a number of baselines such as edge-only scheduling and scheduling with greedy heuristics.

1.4 Structure

We start with describing key concepts necessary for a better understanding of this thesis in Chapter 2. We particularly focus on relevant aspects of the IIoT landscape, serverless computing, and reinforcement learning. Additionally, we discuss related work which addresses similar problems in Chapter 3. We then dive into the main contributions of this thesis. We first present the design of our system architecture in Chapter 4. As a major component of our architecture is our DRL-based scheduler for serverless functions, we devote Chapter 5 to describe its design and internal workings in detail. Chapter 6 elaborates on the implementation choices we made. We then present an extensive experimental evaluation of our approach in Chapter 7. In the first part of this chapter, we conduct experiments to evaluate the *performance* of our DRL-based scheduling mechanisms in isolation. In the second part, we evaluate our system end-to-end, by applying it to a realistic IIoT use case from the light metal manufacturing domain, demonstrating its *feasibility and industrial relevance*, and aiming to identify potential bottlenecks. We conclude this thesis with a summary of our key findings and contributions, as well as a discussion on avenues for future work in Chapter 8.

CHAPTER 2

Background

This thesis explores the application of Reinforcement Learning (RL) to optimize resource allocation and enhance performance in IIoT edge systems. As such, it addresses architectural, system and algorithmic aspects. Specifically, we focus on the challenge of efficiently scheduling serverless functions, such as machine learning predictions, in real-time. Given the dynamic and unpredictable nature of IIoT environments, where data volumes and processing demands can fluctuate significantly, traditional scheduling approaches often fall short. This is where (D)RL, with its capabilities, offers a powerful solution. By enabling an agent to learn optimal scheduling policies through interaction with the environment, RL can adapt to changing conditions and improve overall system efficiency.

This background section provides a foundational overview of the key concepts and technologies relevant to this research, including IIoT systems, serverless computing, and the fundamentals of reinforcement learning, setting the stage for a deeper exploration of their integration and the resulting performance benefits.

2.1 Industry 4.0 Specifics

2.1.1 Industrial Internet of Things (IIoT)

A foundational understanding of the Industrial Internet of Things [SSH+18] is crucial to contextualize this research, as it forms the operational environment, including the underlying infrastructure, for the use cases we target. To start with, the Internet of Things (IoT) paradigm is broad and – seen from an architectural perspective in the scope of the computing continuum [DCD23, AJJ+24] – includes various devices within the edge layer. Specifically, these are devices on the lowest end, often of small form factor, such as mobile phones, smart watches, sensors, but also generalizing to include any other device capable of exchanging data with a network, such as terrestrial and aerial

vehicular platforms or sensor-equipped networked machinery. We particularly speak of IIoT within the scope of industrial applications as in factories. Therefore, IIoT defines infrastructure where IoT devices such as machine sensors, controllers and other devices within the industrial environment can be easily integrated and interconnected (often via wireless technologies [ZWZ+25]) which results in a smart system, improved automation and efficiency.

2.1.2 OPC UA

A key component of data exchange in IIoT is the OPC Unified Architecture (OPC UA)[1] standard, which provides a robust and secure mechanism for industrial communication, enabling interoperability between devices and systems from different vendors which includes communication from machine-to-machine, machine-to-enterprise and everything in-between. OPC UA is seeing significant industrial adoption worldwide [OPC24]. The OPC UA protocol employs an information model to define a structured and hierarchical representation of system data. This model is organized around a root node, from where the data nodes are accessible. Subordinate to the root node, the "Objects" node contains instances of various object types, providing a standardized way to organize and access information within the system [GHIU17, Bus24]. In simpler terms, machine sensors use this protocol for data exchange. An OPC UA server contains a tree of *object nodes* which represent sensors in our case. Object nodes contain so called *variable nodes* which store the raw sensor values.

A useful feature for the Industry 4.0 paradigm is the Publish-Subscribe mechanism which was added to OPC UA. With this, applications do not need to request data directly, but instead, clients just subscribe to a server publishing sensor data over a Message Oriented Middleware (MOM) without knowledge of any subscribers. However, by using this mechanism, some sort of information gets lost as subscribers do not know where the data came from. For example, adding context to that data would be highly beneficial in terms of interoperability. This can be done with context brokers employing the NGSI-LD standard for data exchange.

2.1.3 NGSI-LD

NGSI-LD poses a standard information model for data exchange, whose aim is to enhance interoperability. It does so by providing tools which enable the creation of extensions to include domain-specific modelings for example. We envision smart IIoT applications that consume data from multiple sources, such as sensors attached to machines in the factory, databases with time-series data relevant for the manufacturing processes at hand, and other. Such information is referred to as *context* and collectively includes any relevant information about entities, their properties and how these are related, which is consumed by a smart application. For example, in the framework of this thesis, an OPC UA gateway could act as a context provider; interested applications could subscribe to

[1] `https://opcfoundation.org/about/opc-technologies/opc-ua/`

such information via a context broker, and fuse it with context potentially coming from sensors using different data collection standards, as well as data from external sources, without awareness of the underlying connectivity technology.

Precisely, NGSI-LD gives three major benefits. A key advantage is the system's ability to facilitate flexible information discovery and querying. This is achieved through dynamic data discovery capabilities, coupled with built-in query patterns tailored to address common information needs within unbounded, federated information systems. The second thing is the ability to provide context to data of a specific sensor or service, i.e., the standard provides a mechanism to add information about period of validity, geographic constraints, to which machines or factories sensors belong to, and other relevant information. The last point is that NGSI-LD poses a good solution in terms of scalability and alleviates sharing data across distributed systems by providing a developer-friendly interface for data collection [BBFL19].

NGSI-LD represents data in a way that is similar to a connected graph, where information is linked together. This makes it easy to manage relationships between different pieces of data, much like a graph database. Furthermore, NGSI-LD allows applications to update, query, and subscribe to data, making it simple to automatically access information from multiple sources. A reference implementation developed by FIWARE [CSB+19] is the Orion Context Broker,[2] which is also used in this thesis.

2.2 Serverless Computing

2.2.1 Function-as-a-Service

In recent years, the concept of serverless computing started to get a lot of attention in research and application. The most commonly used form is called Function-as-a-Service (FaaS) where code is split up into isolated functions and deployed separately. Often, FaaS is combined with Backend-as-a-Service (BaaS) which provides additional functionality such as authentication, message brokering and persistent storage for serverless applications [SSK+21]. A key advantage of the FaaS paradigm is the abstraction of underlying infrastructure management. This allows users to concentrate on application logic and minimal configuration tasks instead of infrastructure tasks, as the serverless platform takes care of resource provisioning, scaling, and maintenance.

A defining characteristic of FaaS is its inherent elasticity, enabling dynamic scaling. Serverless providers capitalize on this capability by scheduling workloads among available, possibly distributed servers [YIWL21]. Typically, FaaS pricing follows the pay-as-you-go model and major cloud platforms already offer serverless options, such as Google Cloud Functions,[3] AWS Lambda,[4] Microsoft Azure Functions,[5] and IBM Cloud Functions.[6]

[2] https://fiware-orion.readthedocs.io/en/master/
[3] https://cloud.google.com/functions
[4] https://aws.amazon.com/lambda/
[5] https://learn.microsoft.com/en-us/azure/azure-functions/
[6] https://cloud.ibm.com/functions/

The pay-as-you-go model is a consumption-based pricing strategy where users are charged by what and how many resources they use. Unlike subscription-based pricing strategies, where the user is charged recurring fees, fees are calculated on a fine-grained basis. This calculation depends on factors such as number of invocations, each execution's duration, consumed memory, and consumed computation power. It fits perfectly in unpredictable use cases with fluctuating workloads and sporadic usage patterns, thus, saving up on idle resources.

2.2.2 Serverless Function Scheduling

While FaaS scheduling shares certain similarities with conventional web service scheduling for example, it presents new challenges that require novel approaches due to its special characteristics. Unlike conventional web service deployments, where (micro-)service deployments are fixed and are possibly stateful, FaaS functions are stateless and workloads can spike. After execution, these temporary function instances are scaled down and removed from the server to save up resources after some time. This is known as *scaling to zero*. Function instances are typically run in lightweight software containers such as Docker.[7] On subsequent invocations, an instance is recreated which results in a significant initialization overhead known as *cold start latency* [JSSS+19]. This is because necessary function code, software container blueprints, and source code libraries are possibly not cached on that server. In contrast to traditional web services where deployments usually stay in the same place, FaaS function deployments are constantly moving around the server cluster. This shifting happens because, when functions are not being used, FaaS platforms take back the resources they were using – i.e. memory, disk space, and processing power – and give them to other functions that need them. The changing nature of serverless functions makes it hard to come up with a smart scheduling algorithm that considers these unique characteristics. The scheduler needs to consider that functions can run in any server and that resources are freed up and reassigned all the time. Such a scheduler needs to do more than just simply balancing the load like in traditional web services, in order to handle these unique FaaS challenges [YIWL21].

It is important to note that traditionally FaaS scheduling is discussed as a responsibility of the serverless platform, where the scheduler may have access to detailed infrastructure-level information such as the CPU load of different compute nodes. In the settings this thesis addresses, this is not always the case, as the serverless workload scheduler component that our architecture introduces may dispatch functions to different (public or private) FaaS clouds. Particularly in the case of multi-provider serverless, such monitoring information is typically not available externally and the scheduler needs to make decisions only based on metrics it can observe (such as the response times it perceives).

[7] `https://www.docker.com/`

2.2.3 Function Cold Starts

Serverless functions are lightweight by nature, thus, providing good scaling qualities. Despite this fact, providing compute ressources and building containers from scratch is not necessarily a lightweight task. This includes launching a new container, the setup of the function's runtime environment and deploying this function instance. Such initialization processes take up to several hundreds of milliseconds, thus, delaying the execution of the underlying function. This process of launching new function instances is referred as *cold starts*. Wang et al. [WLZ+18] provide some insights on what happens under the hood and how drastic cold starts are. For example, they differentiate the cases based on whether a function is launched on an existing VM or a new one. Consequently, in the latter scenario, the cold start latency is further increased. Other factors such as memory and language influence the cold start latency additionally. For AWS, they measured median cold start latencies of 167-171 ms for Python 2.7 and 824–974 ms for Java environments. They used a Nodejs 6 environment as baseline to measure cold start latencies which resulted in median values of 250-265 ms for AWS, 110-493 ms for Google and Azure 3640 ms (the reason here was that each instance was assigned 1,5GB).

Upon invocation, a function instance does not immediately terminate but remains active for a brief period. This approach, known as a *warm start*, conserves resources by allowing the instance to handle subsequent requests arriving within a short timeframe, thus avoiding the overhead of repeated initialization. Consequently, warm starts enhance performance by reusing existing, readily available function instances at the expense of increased resource consumption.

The problem with cold starts is the low reliability in critical use cases where performance is important, such as in industrial settings with real-time requirements. Especially in cases where cold starts take up to 3-4 seconds, as Wang et al. have measured [WLZ+18], this poses a significant problem.

Cui [Cui18] stated an interesting edge case which makes this problem even worse. When multiple requests occur concurrently so that multiple function instances are launched, multiple concurrent cold starts can happen which in sum takes more time than just waiting for a single instance to launch where all functions could be invoked. Mohan et al. [MSD+19] also discuss that issue. Such scenarios commonly occur in large systems and especially during periods of time where exceptional traffic is expected, such as rush hours in systems like Uber.

2.3 Reinforcement Learning

Reinforcement Learning (RL) has gained massive attention in the last years generally, and is also being applied in the field of serverless computing [ARB21, FR24, YIWL21]. The general idea comes from natural situations people or animals encounter. In various situations we encounter problems we try to solve by interaction with a specific environment. For example, newborns know nothing about how the world works but are forced to interact

in different environments like how to get from one point to another more efficiently. By taking action they constantly gather new knowledge from how the environment changes which helps them to learn. Additionally, each action results in some kind of feedback which is an indication of its quality, resulting in a positive or negative reward. Quoting Sutton and Barto, whose work [SB20] was a major source for this section:

> "Learning from interaction is a foundational idea underlying nearly all theories of learning and intelligence."

In short, Reinforcement Learning is a strategy to map situations or environments to specific actions, where the goal is to maximize the reward by reaching a goal in the most efficient way, basically a trial-and-error strategy.

The problem of RL is framed using ideas from dynamical systems theory. Precisely, it is described by finding the best way to control a Markov Decision Process (MDP) where not all information is known. A decision is made by obtaining the most relevant information of the real problem (e.g. from the environment) where a learning agent learns the best actions given this information to accomplish an objective. This objective must be related to the observation or state of the environment and an action must have the capability of changing the environment. Consequently, in its simplest form, a MDP is characterized by an environment, an action and a goal. Reinforcement Learning is any method which is able to operate within this MDP problem statement [SB20].

The most used form of machine learning is *supervised learning*, where an external oracle is used to give information and context about data. That is, given a specific situation, an example is labeled which gives information on how to act in that situation. In typical cases such labels categorize examples. The primary goal of this type of learning is for the system to generalize its responses, enabling it to perform accurately in situations that were not encountered during training. While this type of learning is valuable, it is insufficient for scenarios involving learning from interaction. In such interactive settings, acquiring a comprehensive set of training examples that accurately represent all possible situations the agent may encounter is often infeasible. Particularly in novel or unexplored environments, such as in the serverless field and FaaS, where the benefits of autonomous learning are important, an agent needs the capability to learn autonomously from its own experiences.

The RL paradigm does not fit into the term *unsupervised learning* either. Unsupervised learning works by finding structure or patterns among unlabeled data. The key difference between RL and unsupervised learning is that instead of finding patterns, RL aims to maximize rewards from actions to achieve a specific goal based on a specific state of the environment.

By the definition of RL, an agent tries to maximize rewards by taking the best action given a state of the environment. By continuously learning, the agent reaches a point where it learns about the best actions and continues taking those actions for future

iterations. The problem here is that the agent gets biased towards those actions possibly missing better fitting options that the agent has not considered yet due to exploiting his knowledge of past experiences. So, in order to function the best possible way, an agent needs to know every actions reward given an observation of the environment, thus the agent needs to explore different actions. Multiple executions of each action are required to obtain a reliable estimate of its expected reward. Concretely, a big challenge of RL is the trade-off between *exploitation* and *exploration*. The challenge lies in the fact that neither exploration nor exploitation can be pursued in isolation without hindering successful completion of the goal. The agent must balance trying a range of actions with progressively favoring those that yield the most promising results.

2.3.1 Policies and Value Functions

By the nature of RL, given a specific state, the algorithm decides on that state how good each action is in terms of future reward. Specifically, a *value function* on a state-action pair tells how good it is to execute that action within that state. How good the future reward is depends on the chain of actions taken. Therefore, value functions are defined in relation to a specific policy, which is the strategy the agent uses to select actions.

> "Formally, a policy is a mapping from states to probabilities of selecting each possible action. If the agent is following policy π at time t, then π(a|s) is the probability that At = a if St = s." [SB20]

2.3.2 Returns and Episodes

Basically, the goal of a RL agent is to maximize its cumulative rewards over time. In other words, it tries to maximize the expected return, which is defined in its most basic form as a specific function of the sequence of rewards, e.g. the sum of rewards. In applications featuring a natural endpoint or terminal state, where the agent-environment interaction naturally segments into discrete subsequences known as episodes, this approach is suitable. An episode ends when a terminal state is reached. For example, when playing a game, each action is tied with a specific reward. A terminal state occurs, when the game is won, e.g. the goal of that game was reached, or when the game is lost. In other words, when a state is reached where no more actions can be made. The next episode would then be the next round of a game, a reset of the environment, independent of what happened in the previous episode.

2.3.3 Exploration vs. Exploitation

As mentioned before, finding a good balance between *exploitation* and *exploration* is hard. A simple and pragmatic approach is the ϵ-greedy method. It uses a fixed ratio between exploration and exploitation, choosing the action with the highest value but having a chance ϵ of exploring by picking a random action. For example, if the value of ϵ is 0.2,

then there is a chance of 80% of choosing the best fitting action and a 20% chance of choosing a function randomly.

This is called the exploration-exploitation trade-off. This trade-off is central to reinforcement learning, influencing both the certainty of the outcome and the rate at which this certainty can be enhanced and variance reduced. Other exploration strategies also exist. For example, another technique is the ϵ-ratio which starts with a specific ratio but changes over time depending on the learning process [Pla22]. For a thorough survey of different exploration strategies in RL, the reader is referred to the work of Amin et al. [AGS+21].

2.3.4 Q-Learning

A widely used Reinforcement Learning method is Q-Learning [WD92]. This model-free algorithm enables agents to learn about the optimal action-selection policies in Markov Decision Processes. In its basic form, it is defined by

$$Q(S_t, A_t) \leftarrow Q(S_t, A_t) + \alpha[R_{t+1} + \gamma maxQ(S_{t+1}, a) - Q(S_t, A_t)]. \tag{2.1}$$

The so-called Q-values represent a numerical estimate of the expected future reward that an agent can obtain by taking that action and following the optimal action-value policy after. This equation represents how the Q-value is composed. When choosing the action, the new Q-value gets updated by having the current one as base, adding the immediate reward and the next states maximal Q-value but subtracting the current one again, dictated by the learning rate to create a blend of the old estimate and the new information. The components of the above expression are as follows:

- $Q(S_t, A_t)$ represents the Q-value by taking action A_t in state S_t at time t.
- α is the *learning rate* which dictates how fast the algorithm updates its estimates given new information. The value ranges between 0 and 1.
- R_{t+1} is the immediate reward after taking action.
- γ represents the *discount factor* ranging between 0 and 1 controlling how much importance to give the future reward.
- $maxQ(S_{t+1}, a)$ represents the maximum of the next states S_{t+1} Q-values for all possible actions.

In this equation, the Q-value directly approximates the optimal action-value function, independent of the policy being followed. This simplification greatly facilitated the algorithm's analysis and enabled early convergence proofs. While the policy still influences the visitation and updating of state-action pairs, the sufficient condition for guaranteed convergence is that all pairs continue to be updated [SB20].

Q-learning is an *off-policy* algorithm. Off-policy learning methods differ from on-policy methods in how they update value estimates. In off-policy learning, the update is based on the value of an action that was not selected by the current behavior policy. This is particularly useful during exploration, when the behavior policy deliberately chooses a potentially non-optimal action. While an on-policy method would update based on this exploratory action's likely lower value, an off-policy method can instead update based on the estimated value of the optimal action, in case of Q-learning, the best value of the next state $maxQ(S_{t+1}, a)$. This allows off-policy learning to avoid incorporating potentially misleading values from exploratory actions into its updates, thus maintaining a more accurate value function, while also not influencing the behavior policy. So, it allows the agent to learn the optimal policy even while exploring using a different policy.

An important difference between off- and on-policy methods is convergence. This means that after enough iterations, the algorithm recognizes optimal Q-values for all actions. Off-policy methods converge when learning from greedy rewards after sufficient iterations. ϵ-greedy on-policy methods do not converge on the other hand as they keep exploring except in the case of a decreasing ϵ value [Pla22].

Bellman Equation

The value function (i.e., how Q-values are updated) for Q-learning is based on the Bellman Equation. The roots of the bellman equation lie in the idea of getting information of the whole environment. For instance, when looking at the whole state space as a tree-like structure, the naive approach of learning the optimal path would be to traverse the whole tree. Exactly this is the main idea of Bellman, having knowledge of the whole environment at once. So, when looking at a specific state-action pair, the optimal policy is calculated by looking at all possible subsequent states recursively. The Bellman Equation represents the relationship between value function in state s and the subsequent state s', when following the transition function [Pla22].

$$V^{\pi}(s) = \sum_{a \in A} \pi(a|s) \left[\sum_{s' \in S} T_a(s, s') \left[R_a(s, s') + \gamma \cdot V^{\pi}(s') \right] \right], \tag{2.2}$$

where:

- $\pi(a|s)$ is the probability of action a being chosen in state s.
- T_a is the transition function for action a.
- R is the reward for action a transitioning from s to s'.
- γ is the discount factor.

At the end, the same value function is called again recursively on the subsequent state $V^{\pi}(s')$ following the policy π, thus taking the whole state space in consideration. This method falls into the *dynamic programming* paradigm introduced by Bellman.

2.3.5 Deep-Q-Networks

As described earlier, simple Q-learning has a major drawback with its state space explosion problem and is therefore only usable for simple environments with small state spaces. In classical RL scenarios, consecutive states are often very similar to each other. This leads to biased updates, instability as the algorithm can get stuck in local optima, and slow convergence due to redundant information in correlated experiences. The Deep Q-Network (DQN) approach [MKS+15] aims to break correlations between subsequent states and to slow down changes to parameters during training to improve stability, which is achieved using an *experience replay* and slowing down weight updates [Pla22]. Minh et al. found a solution to this by incorporating Deep Learning into Q-learning, as described in their seminal DQN article [MKS+15]. Therefore, key features that distinguish DQNs are the following.

1. Deep Neural Networks (DNN) which approximate Q-values instead of Q-tables.
2. A replay buffer stores past experiences which are used to update the neural network.
3. A target network serves as separate neural network to calculate the target Q-values according to the Bellman equation.

Deep Neural Networks are utilized to approximate the Q-function which estimates the value of taking a specific action in a given state. The state parameters are mapped to the input layer of the DNN. It then learns the complex relationships between these metrics within the hidden layers and provides the results in its output layer, which is mapped to the resulting actions. The DQN agent selects the action with the highest predicted Q-value, effectively learning to optimize a given problem.

Experience Replay Buffer

A challenge in reinforcement learning arises from sequential experiences, as preceding states are strongly correlated to its subsequent states. This can lead to an imbalance in training, where the agent overemphasizes certain areas of the state space while neglecting others, thus lacking in exploration. Additionally, the use of function approximation and bootstrapping can result in the agent "forgetting" previously learned behaviors. For instance, upon reaching a new and different level in a game, an agent might lose proficiency in earlier levels due to this overemphasis on recent experiences.

In order to mitigate these correlations, a *replay buffer* is used. This buffer contains *experiences* which are quadruples consisting of (state, action, reward, next state). It caches a specific amount N previously reached states, which are randomly sampled in the training process. This mitigates the issue of correlated training samples by employing a buffer that stores a dynamic dataset of recent interactions. Instead of training solely on consecutive states, samples are randomly drawn from this buffer, making the training set more diverse. This approach increases the independence of subsequent training examples,

as the next state to be trained on is not necessarily a direct successor of the current state, but rather one selected from a history of past states. Consequently, experience replay reduces temporal correlations between samples by distributing learning across a wider range of previously encountered states [MKS+15].

In the DQN algorithm all examples of the replay buffer are treated equally in the base algorithm. There also exists an approach where sampling occurs according to relevance [SQAS16].

Target Network

The second upgrade which made DQN possible are *infrequent weight updates*, as Minh et al. state [MKS+15]. In traditional tabular Q-learning, the Q-values are updated after every single interaction with the environment. This means that each time the agent takes an action and observes the next state and reward, the corresponding entry in the Q-table is immediately updated. In DNNs these immediate updates can cause instability and correlations between states. Instability occurs when the neural network's weights fluctuate excessively during training, hindering the learning process and preventing the network from converging to an optimal solution. This can happen due to frequent updates based on correlated experiences or noisy data, leading to oscillations, divergence, or getting trapped in local optima.

So, the reason behind the target network is that the Q-network is periodically cloned to create a target network. This target network is then used to generate the Q-learning targets for the subsequent n updates to the main Q-network. In the original DQN, a single network with weights was used, resulting in a constantly shifting target for the loss function. By contrast, the target network's weights are updated less frequently, leading to slower changes in the target values compared to the main network's updates. Using a separate target network addresses a key challenge in Q-learning, where an update immediately alters the target value at each step, potentially causing oscillations or divergence of the policy. Introducing a delay, by generating targets with an older set of parameters, mitigates the risk of these oscillations [Pla22].

CHAPTER 3

Related Work

While IIoT, Industry 4.0, serverless, and edge computing have received a lot of attention, works that particularly apply serverless in concrete IIoT settings and provide architectures or systems combining these elements are scarce. We discuss scheduling of such serverless functions using Reinforcement Learning, where we come up with a different technique in opposition to similar work done in that field. In the following, we provide a short overview of such works and outline the features that set this thesis apart.

3.1 Industry 4.0, Industrial IoT and Edge Computing

Dobaj et al. [DIKK18] propose a microservice-based software architecture tailored for Industrial IoT (IIoT) environments. They argue that the flexibility, scalability, and maintainability of microservices can address the heterogeneous nature of IIoT devices and protocols. However, they rather propose a very high-level IIoT design and focus on a microservice architecture. Their main arguing point are interoperability challenges between devices and systems in this industry. *Our architecture design concretely addresses these interoperability challenges, while focusing on the evolution towards the serverless paradigm.*

Another similar approach is proposed by Hussain and Salehi [HS24] where a dynamic fog federation approach is proposed to overcome the limitations of individual resource-constrained fog systems. Their goal is to optimally and without human intervention partition microservice-based applications across the federated fog in the Industry 4.0 space. The idea emerges from the problem that fog systems are resource-limited, inelastic, disaster-prone, and without access to the cloud. Similar to this thesis, they explore possibilities of serverless within Industry 4.0. However, *they focus on a problem orthogonal to ours*, that of dynamic fog resource federation when there is an increased need for compute resources that can be covered by opportunistically deployed mobile infrastructure (e.g., on drones deployed in emergency response scenarios).

Golec et at. [GGW+24] study the use of serverless computing for predictive maintenance tasks in IIoT scenarios. They focus specifically on the problem of cold start latencies, which is well-known in serverless [LGC+22]. Particularly, they apply machine learning tools to predict cold start delays, which can be a first step toward cold start latency mitigation strategies. While their system includes components that span the whole computing continuum, *the serverless platform where functions are executed resides in the cloud*, and Google Cloud Functions are used. Furthermore, the scope of their work is *limited to cold start latency prediction.* Other works have also proposed ways to reduce the effect of cold starts, either by means of prediction and pre-warming to make slow cold starts rare [KS24, FSP+24], or by using lightweight runtime environments for function execution to make cold starts cheap [GFD22, MPB24].

Simion et al. [SWlT+23] present a framework for "seamless serverless computing across an edge-cloud continuum," directly addressing the need for efficient and flexible deployment of serverless functions in both edge and cloud environments. They introduce a novel edge orchestration technique that enables functions to be deployed and managed seamlessly across the continuum on top of Knative. While this closely aligns to our investigation and gives insights to our idea of deploying functions on both cloud and edge nodes, our aim is a different one. As mentioned before, *we explore the integration of serverless to a lower level* with a focus on performance.

Bacchiani et al. [BPS+22] report on an IIoT use case that, albeit on a different field, has some striking similarities with the one we consider, such as the use of compute resources across the computing continuum to support low-latency processing of streams of data originating by industrial machinery and, at the network level, collected via common protocols (OPC UA data acquisition; W3C WoT to address interoperability). In particular, anomaly detection algorithms are executed on data from CNC or PLC machines on compute nodes at the edge and in the cloud. Contrary to this thesis, and use-case specifics aside, *the granularity the authors consider is that of the (micro-)service and not the individual function as in FaaS, and they rely on a cloud-based orchestrator, which is contrary to our design goals.*

Finally, the work of Baresi et al. [BQT23] is relevant for this thesis as they recognize challenges of resource allocation for serverless functions in edge environments, specifically highlighting how existing approaches often neglect dependencies between functions. We do not address this in this work, but this might be relevant for future work and give some insights of how to further optimize function calls at the edge. They extend and improve an existing framework that considers function dependencies (i.e. how functions are invoked by others) into the new one, NEPTUNE+. Notably, Xie et al. [XGT+23] also consider inter-dependent functions by proposing a reinforcement learning-based serverless workflow scheduling scheme for IIoT services in edge environments. These works underscore the importance of considering dependencies in resource allocation for serverless edge computing and provide *valuable reference points for further research in this area.*

3.2 Reinforcement Learning for Serverless Computing

Li and Nastic [LN24] recently came up with an approach parallel to ours, where a multi-agent reinforcement learning is used for eventually offloading functions from resource-constrained edge nodes, e.g. mobile devices, to cloud providers in order to minimize the completion latency. An agent runs within an edge device and communicates with other edge agents over a *communication group* to ensure that offloading decisions are globally optimal by exchanging relevant data like computing power and network properties. As not all information is necessary, they introduce an *attention* mechanism to decide which information are relevant for other edge nodes to be shared. This reduces communication resources and improves decision making. They put a focus on communication strategies between agents. *In contrast, our work is about a single-agent DQN approach which runs as a separate application observing all available serverless function providers, then scheduling the function to the best fitting instance based on the current state of all instances. Moreover, we observe the scheduling problem in an industrial setting where the edge represent an on-premise server instead of resource-constrained mobile devices.*

An interesting application of Reinforcement Learning (simple Q-Learning) is used by Agarwal et al. [ARB21] to tackle the problem of cold starts for serverless functions. Their idea is to predict function invocations by considering the number of function instances and their CPU utilization for the state. Depending on a specified CPU utilization range, function instances are proactively scaled up or down, thus being able to circumvent cold starts. *This work takes cold starts into consideration as well but from a different perspective. Instead of scaling function instances appropriately, our DQN decides at the application level to which serverless provider a function call should be scheduled.*

Furthermore, we are using DQNs instead of simple Q-Learning to avoid the state space explosion problem and our scheduler offers greater flexibility by operating at the application level, enabling fine-grained control over various aspects of serverless execution, such as the prioritization of minimizing cold starts, minimizing latencies or decide where to run this function based on its type. Femminella et al. [FR24] also discuss the problem of cold starts and scaling of serverless functions at the edge. Their work is also related to ours and both deal with the same technologies. However, they mirror the work of Agarwal et al. [ARB21] but with a more advanced procedure of using different Deep Reinforcement Learning algorithms instead of simple Q-Learning. Additionally, they compare these algorithms in terms of performance and reliability of results, precisely DQN, PPO, and A2C. As other DRL algorithms might be suitable for a function scheduling problem, we decided on DQN due to its stability by reducing correlations between updates, its simpler implementation and better sample efficiency as DQN uses a replay buffer of past experience for better sample efficiency. However, a potential limitation of our approach in certain scenarios is the discrete action space which means that our approach works only for a fixed set of actions whereas PPO for example supports continuous action spaces.

Yu et al. [YIWL21] introduce a novel scheduler employing DRL which considers the changing nature of serverless functions. Implemented in Openwhisk, they replace the

built-in scheduler with their own and compare its performance with baseline schedulers. *Our work sets a focus on black-box (though not limited to) information as our scheduler runs outside of serverless providers like Openwhisk.* They also point out why more sophisticated scheduling methods are required for serverless functions.

In our previous work [Sch22], we proposed a tabular Q-Learning approach for the same problem of function scheduling for general-purpose FaaS instead of IIoT scenarios. The major drawback of that approach was the state space explosion problem, which we avoid by using DQNs in this work and explore this problem in more depth with a broader state representation.

CHAPTER 4

System Design

4.1 Overview

In this section we briefly describe the system design without implementation-specific details, which is depicted in Figure 4.1. For the proposed architecture, we keep generality in mind regardless of the examined use case. It is designed to fit in any industrial IoT use case which involves diverse machinery containing sensors capable of communicating via the de facto standard communication protocol OPC UA. The system is engineered to efficiently handle high-frequency data streams from multiple sensors, dynamically allocate processing resources, and support manufacturing processes in real-time (such as by executing anomaly detection mechanisms) using serverless functions. This design prioritizes scalability, adaptability, and real-time performance while maintaining a modular structure that allows for easy integration with various other IIoT applications and components. By leveraging the standardized data representation of NGSI-LD and the flexibility of serverless computing, this architecture offers a robust and versatile solution for diverse industrial settings.

4.2 OPC UA Gateway

The role of the gateway is to be able to collect diverse sensor data over the OPC UA protocol. It acts as a bridge between the OPC UA servers embedded in manufacturing machines and the central context broker. The gateway subscribes to relevant OPC UA nodes, receives real-time data updates, and transforms the data into a format suitable for the context broker. This ensures interoperability between the diverse sensor data sources and the standardized data representation used within the system. Additionally, the gateway can perform data preprocessing tasks, such as filtering, aggregation, or normalization, before forwarding the data to the context broker. This reduces the data volume transmitted over the network and prepares the data for efficient processing by

downstream components. It is designed to accept various sensor data types which can occur.

4.3 NGSI-LD Context Broker

The context broker serves as a central point for data collection and persistence. Data can arrive from any gateway or data source such as the OPC UA gateway, which we consider as the main data source. It uses the NGSI-LD standard, a JSON-LD based format that provides a rich and standardized way to represent entities and their relationships. This ensures interoperability and facilitates data sharing among different components and applications within the IIoT system. In an Industry 4.0 architecture, adding context to data is crucial to enable extensibility of the system. Instead of applications handling raw data directly, an intermediate context broker adds comprehensive information to data about the origin or for what this data should be used for example. This provides an immense benefit for applications using these data as information about data values is always present. The context broker not only stores the data but also enables efficient querying and retrieval, allowing authorized applications to access the information they need. Moreover, it offers features like subscriptions and notifications, enabling real-time updates to be pushed to interested parties. By providing a unified and standardized data management layer, the context broker plays a critical role in facilitating data integration The context broker is also designed to efficiently persist data as it comes in, thus, not only being able to add context and forward data, but also saving data into its database. This takes responsibility for storing data from applications, representing a central point of data collection and storing.

4.4 Application Layer

This part does rather represent a layer instead of a single component. It poses a layer of applications listening for data or fetching data from the context broker. It embodies the core functionality and intelligence of the system, leveraging the data provided by the context broker to generate insights, automate actions, or support decision-making. An example is an application collecting and processing data in order to evaluate them using machine learning techniques and make predictions. The applications in this layer can be diverse, ranging from simple data visualization tools to complex machine learning pipelines for predictive maintenance, anomaly detection, monitoring, or process optimization. The modular design of the system allows for flexibility in integrating various applications, enabling the system to adapt to evolving needs and incorporate new functionalities as required.

4.5 Serverless Provider

In our proposed architecture we consider the use of serverless technologies as highly beneficial in terms of resource utilization and extensibility. Specifically, serverless functions may be called by applications in the application layer. In a smart Industry 4.0 system, which generates and processes tons of data, running this in a simple server may cause problems in terms of load and resources. Serverless, especially FaaS, enables efficient resource use as edge resources might be limited. FaaS achieves that by efficient scaling, so that unused resources are freed and provided to other applications. In addition, serverless enables the use of cloud resources easily, as the developer can simply deploy functions on the cloud. This has the advantage that in case of loaded edge instances, function calls can be offloaded to the cloud. Serverless functions provide a high degree of flexibility, enabling the integration of diverse application logic (e.g., detecting anomalies in a light metal manufacturing process, as in the use case we evaluate in Chapter 7) without requiring modifications to the overall system's architecture. This approach aligns with the dynamic nature of IIoT environments, where adaptability and efficient resource utilization are paramount, potentially at the expense of additional latency, especially when using cloud providers.

We should note that we target serverless edge computing [RND23] use cases that involve scenarios where serverless functions can be executed on an on-premise FaaS platform or, when specific conditions are met, be offloaded to external cloud facilities, such as at the edge of the telecom operator's network following the Multi-access Edge Computing paradigm [ETS22] or at traditional remote cloud data centers. We particularly consider *multi-cloud* serverless settings, a paradigm that is receiving increasing attention [BKJS21, ZBPG22]. In our architecture, interfacing with FaaS provider platforms – internal or external – is the responsibility of the *scheduler* component described next, which is in charge of dispatching function invocations to the appropriate serverless providers. As such, the scheduler component implements the integration logic so that function invocation requests are appropriately encoded according to the API specifications of different FaaS providers. In order to support additional FaaS offerings, the appropriate adapters need to be implemented, which is how our design achieves a level of extensibility with respect to serverless provider support.

4.6 Serverless Function Scheduler

In the multi-cloud FaaS scenario we assume, the scheduling problem that we address is positioned at a level higher than what is traditionally the case in serverless. Namely, our scheduler operates *outside* the FaaS platform[1] and decides on a per function call basis which is the most appropriate FaaS provider to use, considering the information

[1]However, the scheduling logic we present in Chapter 5 could be implemented within the serverless platform, where more detailed infrastructure-level monitoring information such as host CPU loads could also be utilized. For such cases, we design and evaluate an appropriate adaptation of our scheduler that considers such load information.

that it can measure (e.g., latency, failure rate) or has otherwise available (e.g., pricing information and availability related SLAs).

Due to the unpredictable nature of serverless functions, we incorporate an intelligent scheduling mechanism into our system. Precisely, applications using serverless functions make use of the scheduler to run their functions according to their specific needs. The scheduler employs a DQN-based decision strategy, which is able to consider specific characteristics of serverless providers and functions, thus, making the best fitting choice for function execution. Example cases, where such a scheduler shines, are loaded edge servers resulting in the cloud being a better location for execution. Another situation would be that functions have tight performance criteria, where the scheduler helps choosing the best fitting provider.

A question, which arose during the design and implementation of our system, is whether this scheduler should be implemented as a single stand-alone component in a separate deployment or whether each application should come with its own scheduler instance. Originally, we opted for the former, so that each application does not have to implement this specific logic. However, applications might have specific needs and priorities resulting in different configurations for the scheduler. To take it a step further, schedulers with fundamentally different logic might be desired to be supported in parallel. Therefore, it might also make sense to run the scheduler as part of the application. While in our implementation and experiments we have eventually focused on the latter approach, bundling the scheduler with the application, we note that our architecture can also be deployed with a single scheduler component with all applications forwarding their function invocations to its API endpoint.

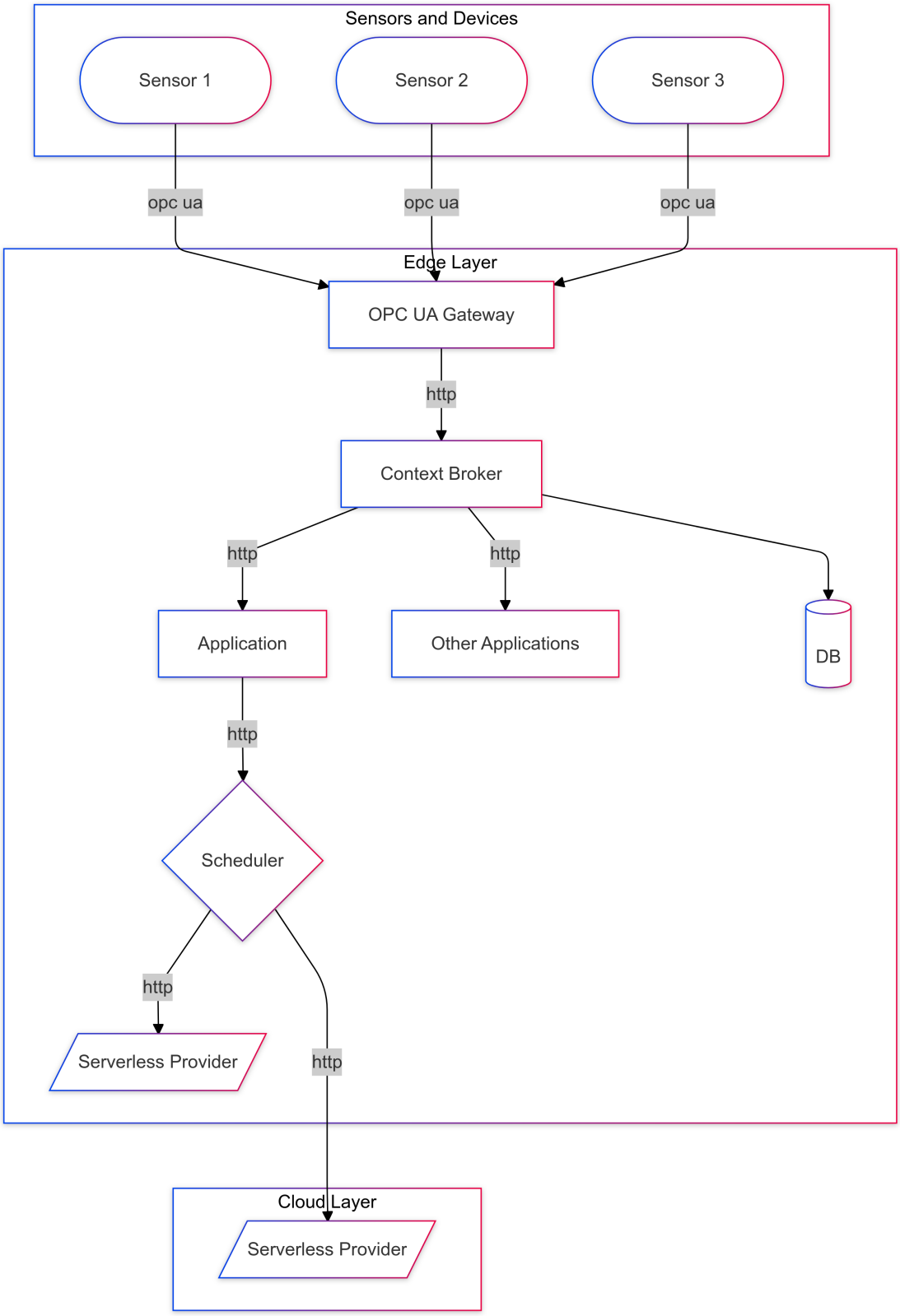

Figure 4.1: Overall System Architecture Design

CHAPTER 5

A Reinforcement Learning-Based Serverless Function Scheduler

In this chapter, we delve into the internals of the serverless function scheduler component of our architecture. We present the design of a scheduler tailored to multi-cloud serverless settings, and extensions that would be possible if the scheduler were deployed in a more controlled environment where it would have access to more detailed platform-level performance information. For each function invocation request, our scheduler is tasked with deciding on the appropriate FaaS platform instance to forward the request to for execution. For this purpose, it implements a decision making strategy based on the DQN reinforcement learning algorithm.

We start from a basic version of our scheduler that aims to optimize for low response times, without sacrificing on reliable function execution. This scheduler operates on minimal observable information, namely the response times, cold start incidents, and invocation failures resulting from its decisions. Then, we propose a DQN-based scheduler design which aims to make decisions that also induce a balanced workload among target serverless platforms by avoiding scheduling on overloaded hosts. This scheduler is tailored to settings where platform CPU load is observable – this is not generally the case when using (external) public FaaS platforms. Finally, we discuss potential extensions that could allow system operators to attach different performance priorities to individual functions, which the scheduler then considers in its decisions. Each scheduler variant naturally comes with a different state representation and reward structure, on which we elaborate in this chapter.

5.1 State Representation

To define the state for the reinforcement learning problem at hand, we first have to think about how a state of a serverless function provider/instance can look like and

which parameters are relevant for function execution. Serverless functions are typically called remotely, so they are not embedded into application code directly, but rather are invoked similar to a remote procedure call (RPC). That means that there is a latency for communication involved. Another factor that contributes to application-perceived response time the function execution time, i.e. how long it takes to produce a response to a called function. Execution time is influenced by different conditions like computation time of the function itself and cold starts. Response time increases significantly in case of a cold start. Those indicators are already a solid measure, but other factors play also a role in a successful invocation, where success can be measured in different ways, for example if a specific response time requirement was met. Next, the actual success of a function, i.e., if it returns a valid response, play a role as retries can be costly in terms of response time. The load of a provider is also relevant, as response times or success rates are affected by that measure, typically by means of CPU-load. However, in many cases this information is not directly observable outside the serverless platform. Lastly, function characteristics and requirements set by the caller are also relevant, e.g., whether the function is computation-heavy, or whether it needs to return within a specific time. Functions may be also labeled according to other needs such as privacy and data locality, where an execution on self-hosted instances is preferred.

The state representation varies between different implementations and grows with the number of providers available. In this work, we evaluated different types of DQN implementations, where each covers different factors in its state. Specifically, the first variant we design includes latencies, cold start information, and availability, which can be a measure for function success, into its state per provider. The second considers CPU-load in addition. Actions correspond to which provider is chosen in a given state.

To clarify the encoding of the state representation, we will talk about the specifics. Each parameter within the state is represented as a normalized numerical value. Latency, for instance, is normalized by dividing the raw latency value by 10 to make the state space smaller. Cold start information is encoded as a binary value, where 1 represents a cold start and 0 indicates a warm start. Availability is an indicator for the function's success. It is encoded by putting it into custom bins, where 99.999% falls into the bin 99.9%. To speed up training convergence in our experiments, we simplified this by representing availability in coarse-grained bins. In the second variant of our scheduler, CPU load is normalized by putting values into 10 bins, each representing a range of 10 for reasons of simplicity. These normalized values are then concatenated into a single vector, where the order of features is consistent across different instances. This vector serves as the input to the DQN, allowing it to learn and make decisions based on the combined information of all relevant factors.

We summarize the state space representation of our two scheduler variants below, where l_i stands for the normalized latency, c_i for the cold start, a_i for the normalized availability and cpu_i for the normalized CPU-load value per instance i.

- **Variant 1** – latency, cold start and availability-aware scheduler:

$$\mathcal{S} = (l_1, cs_1, a_1, \ldots, l_n, cs_n, a_n)$$

- **Variant 2** – latency, cold start, availability and *CPU-load*-aware scheduler:

$$\mathcal{S} = (l_1, cs_1, a_1, cpu_1 \ldots, l_n, cs_n, a_n, cpu_n)$$

5.2 Training and Convergence

Training is a difficult task, especially when it comes to train an algorithm with the aim to suit various situations and patterns possible in the serverless nature. By the nature of Reinforcement Learning, an an agent learns continuously by interacting with the environment by taking action in given states. The tabular Q-learning approach would need to encounter each single situation to learn from it. For larger state spaces training would be close to infeasibility due to the state space explosion problem. DQN's generalize from seen states and approximate Q-values based on that. This reduces training complexity by a major factor compared to tabular Q-learning, but still need to encounter various states and patterns to be able to learn from that. Although, we show good results in our experiments, more training would probably be required with more varying patterns.

We apply the Rainbow DQN approach [HMvH+17], which is supported by RLlib [LLN+18], our DRL implementation framework of choice. The agent trains a Convolutional Neural Network (CNN), as the original DQN [MKS+15] does, with 3 convolutional layers, including 32, 64 and 64 channels respectively. The layers use filters of sizes 8×8, 4×4, and 3×3, with corresponding strides of 4, 2, and 1. It uses a dueling architecture where each hidden layer consists of 512 units. As stated in the background chapter, the output layer corresponds to the actions.

To train it, we simulate various patterns, such as changes in load, latency and computation time, simulating random spikes, in order to provide a solid base. Figure 5.1 and Figure 5.2 depict how convergence looks within 100 training iterations in an example training session of the first variant of our scheduler. The loss function, which is an indicator of how much the optimal reward was missed, gradually decreases until reaching an optimal policy at iteration 42, indicated by a consistently low value. Due to some degree of randomness given by our environment and the random exploitation, convergence can also get reached at a later point. Respectively, the rewards gradually increase and show optimal results given the reward function. We note that rewards show slight peaks or drops due to the nature of the scheduling problem and the dynamic environment. The exploration strategy of the DQN can lead to occasional selection of suboptimal actions, resulting in temporary dips in rewards. However, the overall trend indicates a clear convergence towards an optimal policy, where the scheduler effectively learns to balance various factors and make informed decisions to optimize resource allocation and meet performance objectives.

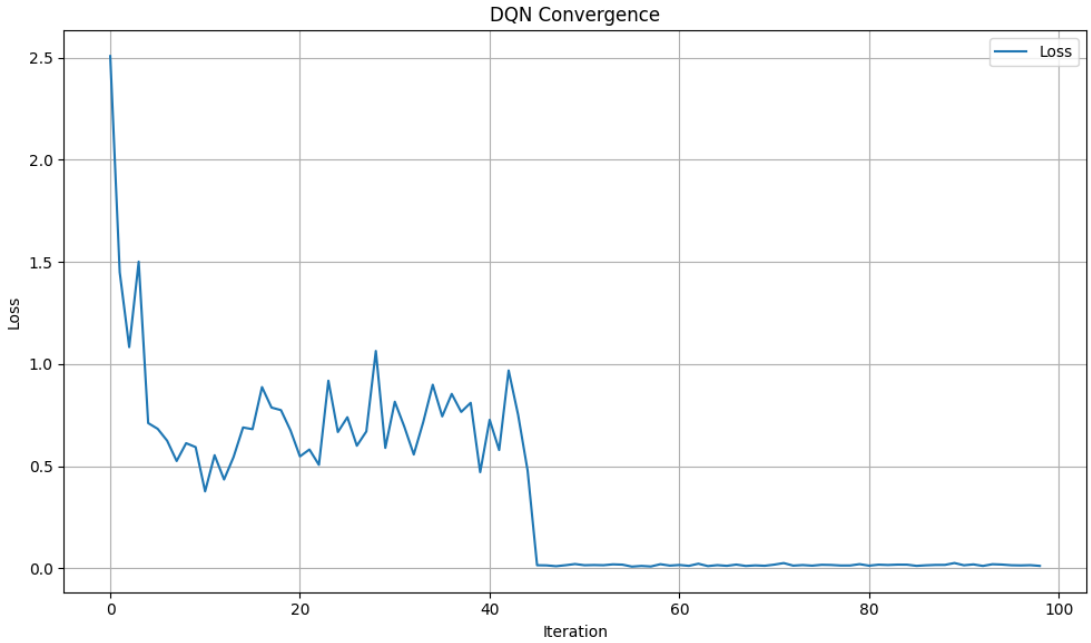

Figure 5.1: Training loss

5.3 Reward Model

5.3.1 Variant 1: Latency, Cold Start and Availability-Aware Scheduler

In the problem of FaaS scheduling, there is no specific goal such as winning conditions in games which indicate if an episode is done or not, we just penalize actions based on their performance. This way we guide the agent towards the most profitable action in terms of latency, cold start avoidance, etc. The following reward functions consider response times, divided by 10, so that they do not outweight failures. Cold start latencies are included within the overall response time, e.g., if a cold start delay increases the response time by 400, paired with a round trip with computation time included, an additional delay of for example 30 would result in a negative reward of -43. Given our focus on mitigating cold starts, function call failures are not heavily penalized. Instead, a moderate negative reward of -40 is assigned. This reward magnitude was determined to be sufficient to incentivize the agent to avoid cold starts and failures effectively while still allowing for exploration and learning in other aspects of the environment, thus providing an initial reward model for evaluation. In the first variant of our scheduler, our reward is therefore given by the following expression:

$$R(t,s) = \begin{cases} -P_f, & \text{if } s = 0 \\ -\frac{t}{10}, & \text{if } s = 1 \end{cases} \tag{5.1}$$

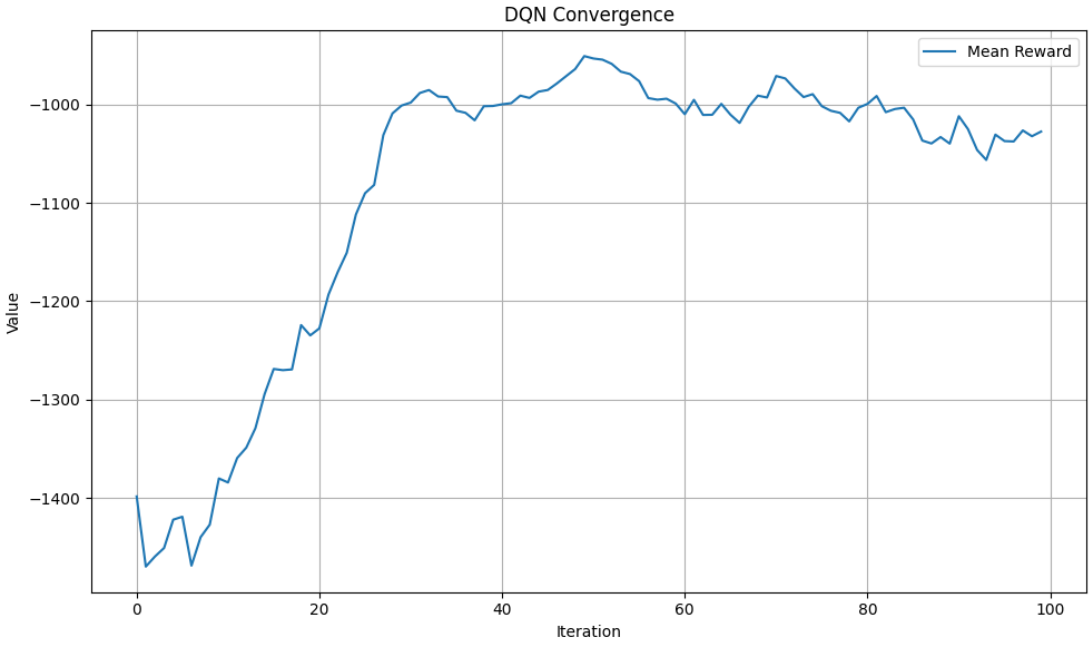

Figure 5.2: Mean rewards per training iteration

- $R(t, s)$ represents the reward function with response time t and success indicator s.
- $-P_f$ represents the penalty in case of a failure.

5.3.2 Variant 2: Latency, Cold Start, Availability and CPU-Load-Aware Scheduler

Additionally, other metrics like CPU-load might be also interesting to observe. Gathering information about CPU load provides a critical parameter for the observation of a serverless function instance's state and overall health. While serverless platforms abstract away much of the underlying infrastructure management, monitoring CPU utilization remains valuable for several reasons. Firstly, CPU-load serves as a key indicator of an instances current processing capacity and its ability to handle incoming requests efficiently. High CPU utilization might signal that an instance is coming closer to its processing limit, potentially leading to increased latency or even request failures.

Secondly, CPU load can be used to proactively predict and prevent performance decrease. In an industrial setting, such decreases can prove as lethal in settings where real-time prediction is required. Manufacturing processes might come with tight performance requirements. Not providing responses within a reasonable time might harm a process significantly and lead to losses. By monitoring CPU load trends, it becomes possible to anticipate when an instance might become overloaded and take preemptive actions, such as provisioning additional instances or rerouting traffic.

Thirdly, in the context of function scheduling, CPU load information can be leveraged to make more informed decisions about where to route new function invocations. For example, a scheduler could prioritize instances with low CPU utilization, ensuring faster execution and minimizing the risk of resource contention.

Lastly, understanding CPU load patterns can be valuable for optimizing function code and resource allocation. By identifying functions or code sections that are particularly CPU-intensive, developers can focus their optimization efforts to improve overall performance and reduce execution costs. In a dynamic and often opaque serverless environment, CPU load emerges as a vital metric for ensuring efficient resource utilization, maintaining application performance, and enabling intelligent scheduling decisions.

Therefore, for scenarios where CPU load information is available to the scheduler, we add an additional factor into the reward function, which penalizes high load. This function was also used for evaluation (see Chapter 7). Even though these reward functions are simple, they were sufficient to show that they work well in the scenarios we studied. In the following reward function that drives the second variant of our scheduler, we consider the variable c additionally, which represents the CPU-load after scheduling a function call to that instance.

$$R(t,s,c) = \begin{cases} -40 - \frac{t}{10}, & \text{if } s = 0 \\ -\frac{t}{10} - \begin{cases} 60, & \text{if } c \geq 95 \\ 40, & \text{if } 90 \leq c < 95 \\ 20, & \text{if } 80 \leq c < 90 \\ 0, & \text{if } c < 80 \end{cases}, & \text{if } s = 1 \end{cases} \tag{5.2}$$

5.4 Extensions

Even though these reward functions perform well in our target scenarios as experiments demonstrate, more sophisticated models are possible and might be useful in other real world settings. Of course, a thorough training with diverse real world patterns is also necessary. We conclude the discussion on the design of our DQN-based scheduler with a possible extension to provide the system some flexibility to efficiently schedule serverless functions with inherently different and potentially conflicting priorities. We factor these priorities in the design of the reward function by introducing configurable weights, each associated with a different criterion (e.g., latency, availability, CPU load), which applications can adapt to their needs more easily.

In particular, we propose a labeling mechanism for functions, which assigns each weight a different value depending on the label. This way it is possible to customize priorities to a certain level on the function level. That means, depending on a label, e.g., “no-fail”, we can schedule incoming function calls in a specific way, such as avoiding failures. The label “no-fail” for example, would give the success penalty the maximum weight factor of 1. Still, the scheduling mechanism is not fully customizable this way. As the function

label would be part of the state, those labels need to be predefined in the system with predefined weights.

A specific reward model for this approach is proposed below. In this model, we utilize a simple linear scaling technique to normalize reward components. This involves subtracting the minimum observed reward and dividing by the difference between the maximum and minimum rewards. This maps the rewards to a range between 0 and 1, ensuring that all reward components contribute proportionally to the overall reward signal.

$$R(t,s,c) = \begin{cases} w_f - w_c \cdot \frac{c}{100} - w_t \cdot t, & \text{if } s = 0 \\ 1 - w_c \cdot \frac{c}{100} - w_t \cdot t - \begin{cases} w_h, & \text{if } c > c_{threshold} \\ 0, & \text{if } c \leq c_{threshold} \end{cases}, & \text{if } s = 1 \end{cases} \tag{5.3}$$

- w_f represents how the penalty for failures is weighted, ranging from 0 to 1.
- w_c represents how the penalty for CPU-load is weighted, ranging from 0 to 1.
- w_t represents how the penalty for response times is weighted, ranging from 0 to 1.
- w_h represents how the penalty for exceeding a defined CPU load threshold is weighted, ranging from 0 to 1.

Thoroughly evaluating these extensions to our basic DQN mechanisms is a subject for future study.

CHAPTER 6

Implementation

In this chapter, we dive deeper and discuss implementation choices and decisions we made in our proposed system. We talk about their advantages and disadvantages and describe the technology used in our implementation. This detailed examination of the implementation process provides valuable insights into the practical considerations and trade-offs involved in building a real-world IIoT system. By understanding the specific technologies, libraries, and design patterns employed, readers can gain a deeper understanding for the complexities of translating a conceptual design into a functional and efficient system. For our deployment scenario, we pack each part in its own Docker container to promote maintainability and extendability.

6.1 OPC UA Gateway

For our gateway, we use the Go programming language.[1] Reasons for this decision are mainly its strength in building concurrent applications such as gateways. The enabling mechanism for this are the so called *goroutines*, which are lightweight threads managed by the Go runtime. In case of our OPC UA gateway, this mechanism enables seamless and efficient processing of multiple sensors in parallel. In contrast to python for example, which we also considered due to its simplicity, python has limited concurrency abilities due to the global interpreter lock preventing multiple threads of executing python code at once.

The gateway starts a subscription within a separate goroutine to a sensor once it receives an http request, thus, serving as a simple web server additionally. It is designed to handle different types of sensors. Therefore, it receives custom subscription details in each request body. Once the gateway subscribed successfully to a sensor, it keeps collecting data and sends data values further to the context broker.

[1] `https://go.dev/`

A major point of discussion is if data should be preprocessed at the gateway level or not. The problem in sending each new value notification to the context broker is a much heavier network load, further increased by the additional context information in a format suitable for the NGSI-LD standard. The preferred solution is to batch values into a list and send the list as update for an entity. This also aligns to the specific use case we evaluate in Chapter 7, where the input of an anomaly detection model corresponds to a list of 10 values. For applications that operate with single values, this approach is still appropriate as single values are extractable with their timestamps. However, it is possible to extend the system by implementing an option whether to send data as a batch or immediately as each value arrives. This makes sense for applications which require data to arrive in a specific frequency. For instance, if a sensor, such as a camera transmitting image data, produces only one data point per second, the resulting update frequency might be too low, especially as the frequency gets lower based on how many items are required to fill the list and trigger the update request.

6.2 NGSI-LD Context Broker

For the context broker, we utilize an existing approach developed by the FIWARE Foundation[2], namely the Orion-LD Context Broker. It employs the NGSI-LD standard [BBFL19] out of the box. It is implemented as a high performance and high throughput middleware and offers various extensions to the system. By default, it uses the document-based MongoDB[3] where entity data is stored. One of its extensions relevant for industrial use cases – and the use case we experiment with in this thesis in particular – is Quantumleap.[4] It is an additional service for storing, querying and retrieving data in NGSI-LD format. Quantumleap converts this data and stores it in a tabular time-series data store, such as CrateDB. An example payload which employs the NGSI-LD format is specified below.

```
{
  "id": "Machine1",
  "type": "Machine",
  "temperature": {
    "value": 23,
    "type": "Float"
  },
  "pressure": {
    "value": 720,
    "type": "Integer"
  }
}
```

[2] https://www.fiware.org/
[3] https://www.mongodb.com/de-de
[4] https://quantumleap.readthedocs.io/en/latest/

6.3 Serverless Layer

For the serverless layer, we utilze Nuclio.[5] Nuclio is a high-performance open-source serverless framework which soothes the lock-in problem, ensures fast development and supports GPU utilization. It is designed for speed and efficiency, making it significantly faster than many alternatives [LKRL19]. Nuclio offers versatile deployment options, running as a standalone Docker container or within a Kubernetes cluster. Nuclio supports a wide range of event triggers, including HTTP requests, message queues, and scheduled events, making it adaptable to various use cases. Its focus on performance, flexibility, and data-centric workloads makes Nuclio a compelling choice for building and deploying serverless functions, suitable for industrial IoT environments.

For our DRL-based function scheduler, instead of implementing the DQN from scratch, we use Ray [MNW+18] and RLlib [LLN+18]. Ray leverages concurrency whereas RLlib provides a full implementation of deep Q-learning as proposed by Minh et al. [MKS+15]. Despite of the vast configuration and customization options provided by RLlib, we use default parameters with little customization which are described in Table 7.2.

[5] `https://nuclio.io/`

CHAPTER 7

Evaluation

In this chapter we present the findings of the experiments we have conducted to evaluate our scheme. The evaluation is twofold. In the first part we measure and evaluate how well our DQN-based function scheduler performs given baselines such as an edge-only heuristic and a latency-greedy heuristic. We further compare two different DQN scheduling algorithms with different state spaces, where we show which benefits a consideration of more state parameters brings. In the experiments dedicated to our scheduler we examine different scenarios serverless functions may encounter. We strive to solve them and outperform our baselines showing the value of our scheduling algorithm in improve the use of serverless in an industrial IoT setting.

For the second part, we evaluate the feasibility and relevance of our approach by deploying and measuring the performance of our system end-to-end on a specific realistic IIoT use case provided by AIT.[1] For this, we simulate the integration by setting up local machine instances, which simulate sensor readings for a light metal manufacturing process. We measure its performance regarding whether our system meets requirements set up by AIT. The major requirement specifies a time limit in which our end-to-end system must deliver. Additionally, we setup a function instance in Nuclio[2] which evaluates sensor readings using an anomaly-detection machine learning model specifically developed for this process by AIT. All experiments are run locally on the same host with the environment specified in Table 7.1.

[1] https://www.ait.ac.at/
[2] https://nuclio.io/

Table 7.1: Experiment Environment Details.

Experiment Environment	
Operating System	Microsoft Windows 11 Home 64-Bit
CPUs	Intel Core i7-10875H @ 2.30GHz, 2304 MHz, 8 cores
Main Memory	16 GB DDR4 RAM
GPU	Nvidia GeForce RTX 2080 Super MAX-Q | 16 GB GDDR5 RAM
Python Version	3.10.13
Torch Version	2.1.1
Ray Version	2.40.0

7.1 Function Scheduling Performance

7.1.1 Emulation Settings

Cold Start Emulation

We emulate cold starts by tracking each function call with the exact time when it was invoked in a map-like data structure per instance. As serverless functions are designed with scalability and resource optimization in mind, function containers are scaled down if not frequently used. A simple heuristic when to scale down functions is time. For different providers this threshold ranges between seconds. Therefore, we assume that we know after how many seconds a function container is stopped/destroyed, which we use to calculate if a function call will result in a cold start or not. For example, if the threshold is 8 seconds, we retrieve the cold start by calculating the difference between the current time and the time this function was last called. However, for our quantitative experiments with several hundred or thousand function calls, we randomize cold start. By doing so, we simulate different possible scenarios over time, thus getting good measures for our experiments.

Serverless Instances

For the emulation of serverless instances, we need to consider latencies and how the instance runs a function. For the latency, we simply preset a realistic range and simulate specific behaviors like random spikes and sudden increases of latency for a short period of time. These latency ranges were gathered by pinging edge and cloud servers. In addition, we emulate CPU-load by increasing and decreasing it depending on function calls. For our experiments, we decide on a fixed computation time each function takes, adding a distribution of the latency range for each instance. Depending on the availability, we emulate failures.

7.1.2 Tools

For all the experiments, we use Ray [MNW+18] and RLLib [LLN+18] to train our algorithm. Ray is a powerful open-source framework that facilitates distributed computing and parallel execution of tasks, making it particularly well-suited for training computationally intensive machine learning models like DQNs. It offers several advantages that can significantly accelerate and streamline the training process. Even when executed on a single computer, it speeds up the training process a lot by exploiting parallelism.

One of the key benefits of Ray is its ability to efficiently distribute the workload across multiple cores or machines, enabling parallel execution of tasks such as experience replay sampling, neural network updates, and environment simulations. This parallelism can lead to substantial speedups in training, especially for complex RL algorithms that require large amounts of data or extensive exploration.

Ray also provides a simple and intuitive API for defining and executing distributed tasks, making it easy to parallelize existing code or implement new distributed algorithms. Its flexible architecture allows for seamless scaling from single-core experiments to large-scale distributed training on clusters, enabling researchers to experiment with different configurations and leverage the full potential of their hardware resources.

Furthermore, Ray's integration with RLlib, a comprehensive library for reinforcement learning, offers a rich set of tools and functionalities specifically designed for training DQNs. RLlib provides optimized implementations of various DQN algorithms, along with features like hyperparameter tuning, checkpointing, and logging, which can further simplify and accelerate the development process.

The agent is driven by the reward function which differs in each use case. So, the goal is rather to provide the best scheduling decision given a specific state considering future rewards as well. For example, after scheduling a function, the agent learns future invocations on the same provider are beneficial as cold starts are probably avoided. This happens due to the nature of DQN considering future rewards for actions too in the current state. That means that the Q-value is calculated including the maximum future Q-value (i.e., in the next state). This way, the agent will learn that a specific action will probably result in a higher reward (i.e., avoiding a cold start) in the next state, thus learning about long-term consequences. During training, the agent employs an epsilon-greedy strategy with a decaying exploration rate to balance exploration and exploitation.

7.1.3 Baselines

In this section, we present the baseline scheduling strategies against which we compare our DQN-based scheduler.

Latency-Greedy Heuristic

A very simple strategy to gain an advantage in terms of performance is to use a heuristic which decides on the instance with the lowest latency value. Although this could be helpful in some scenarios, it is not sufficient to handle the complex nature of serverless functions, but serving as a good baseline for comparison to our smart scheduler.

Function-Failure Heuristic

Similarly, this heuristic selects an instance based on the highest availability rate. If multiple instances share the same value, it follows a latency-greedy approach similar to the latency-greedy heuristic. This serves as our second baseline, where we show that also this heuristic performs significantly worse than our DQN scheduler, which is able to make complex decisions based on multiple parameters.

Edge-Only Heuristic

A common strategy of function execution or computation in general would be to use existing on-premise edge infrastructure due to potential cost savings or because of security and privacy constraints. This approach can simplify management, reduce dependencies on external providers, and potentially offer greater control over data security and privacy. Therefore, we take this as our third baseline.

7.1.4 DQN Evaluation: Latency, Cold Start Calculation, Availability

We start with a DQN agent, which is trained for optimizing function scheduling regarding mean instance latencies, cold start executions and availability rates. We consider a weighted mean of the last three latency values which were measured per instance.

$$\overline{L_i} = 0.1l_i^{(3)} + 0.3l_i^{(2)} + 0.6l_i^{(1)}$$

We choose this approach as the latest latency value might not be representative and would bias the state representation towards random spikes. By using a weighted mean of the last three values, we get more stable information about the latency of an instance which is more resistant to sudden latency spikes. Furthermore, we want to capture most recent data of an instance which is why we calculate the weighted mean, where the most recent latency measure gets the most weight for the resulting mean, i.e. 0.6.

This DQN agent is trained with 1000 iterations, where each iteration contains 32 batches (or timesteps), each running 50 episodes. Since our environment does not specify a specific goal to reach, we just limit an episode with 50 steps. 1000 training iterations was found to be a good starting point for the agent to reach a good level of convergence. This batch size (which is the default value) and episode length were chosen to balance computational efficiency with sufficient exploration of the serverless environment's state and action space.

Table 7.2: Summary of DQN-parameters.

Steps per Episode	**Training Iterations**	**Batch Size**
50	1000	32
Replay Buffer α	**Replay Buffer** β	**Replay Buffer Capacity**
0.5	0.5	60 000
Learning Rate α	**Discount Factor** γ	
0.001	0.99	

Comparison of our DQN scheduler with a Latency-greedy and Edge-only Heuristic

In this experiment, we conduct a comparative analysis of our DQN-based function scheduler against two baseline heuristics: a latency-greedy heuristic and an edge-only heuristic. The latency-greedy heuristic selects the serverless provider with the lowest expected latency at the time of the function call, without considering cold starts or availability. The edge-only heuristic exclusively utilizes edge instances for all function executions. We evaluate these strategies using simulated instances, focusing on average latency and cold start frequency as our primary performance metrics. This analysis aims to demonstrate the effectiveness of our DQN scheduler in optimizing function placement decisions and improving overall system performance compared to simpler, reactive heuristics.

Each scheduling method receives the exact same function to call and the same observation every round. The number of function calls is 1000. We provide a set of three different functions. For the sake of simplicity, cold starts are calculated randomly. The experiment is run with the instance configuration shown in Table 7.3.

Table 7.3: Instance Settings.

Instance Type	**Availability**	**Latency (ms)**
Edge	99%	5-15
Private Cloud	99%	16-25
Public Cloud	99%	25-40

The result shows that our DQN-scheduler outperforms the other strategies when it comes to cold starts. We simulate cold starts by adding a gaussian-distributed latency of 300-500 ms, which results in almost no response times around 200 ms. Most of the response times lie between 5 and 100, where the median lies on the lower end as the diagram shows. The diagram especially shows that cold starts were effectively avoided as the frequency of response times around 400 is significantly lower compared to the other strategies and as the interquartile range shows (the thicker line), most of the datapoints lie in range

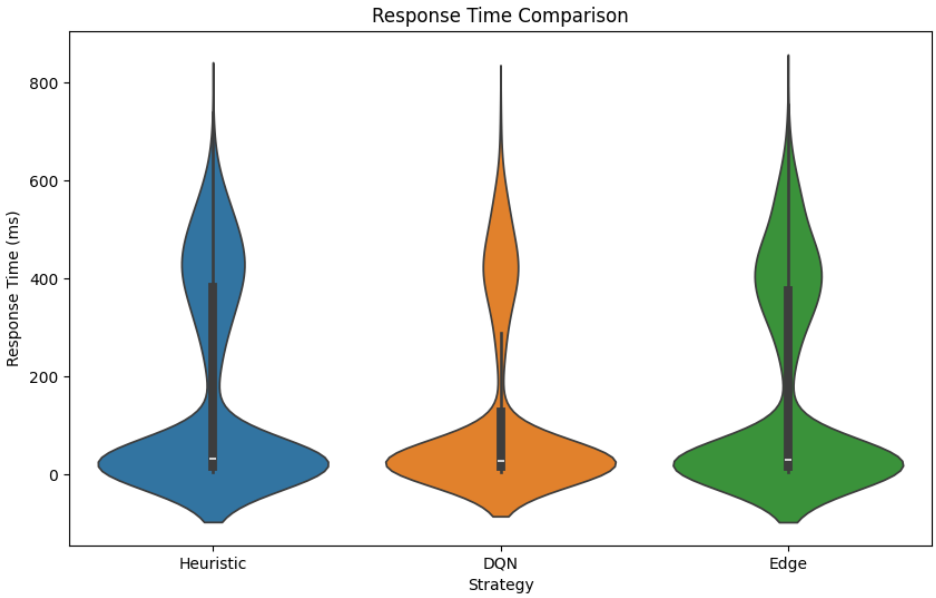

Figure 7.1: Violin chart representing response times and frequencies.

0-100 whereas the interquartile range of the other strategies reaches up to around 380.

Comparison of the DQN Scheduler and the Latency-Greedy Heuristic for Function Batches

Next, we will show how consecutive calls of the same function are handled by the DQN strategy and the latency-greedy heuristic. We send three batches of the same function 15 times to our scheduling mechanisms, which consist of our latency-greedy heuristic and our DQN-scheduler. With that we want to show how both strategies handle this type of workload. To do this, we configured the three available instances to have the same latency ranges and the same availability. We do this to put a focus on cold start avoidance so that the decisions are not biased from availability or latency. We solely want to show the decision-making capability with respect to cold starts.

As Figure 7.2 shows, after scheduling a function call to the best fitting instance regarding latency and availability, while a cold start is unavoidable in all available instances, our algorithm effectively avoids cold starts for consecutive function calls. In contrast to the heuristic, a cold start occurs only on the very first invocation of the function. Due to the latencies being similar to each other, the latency-greedy heuristic simply chooses the instance with the lowest latency, thus, leading to eventually schedule the call to all instances after some time and an overall higher cold start rate.

Our DQN scheduler demonstrated its effectiveness in minimizing cold starts during the

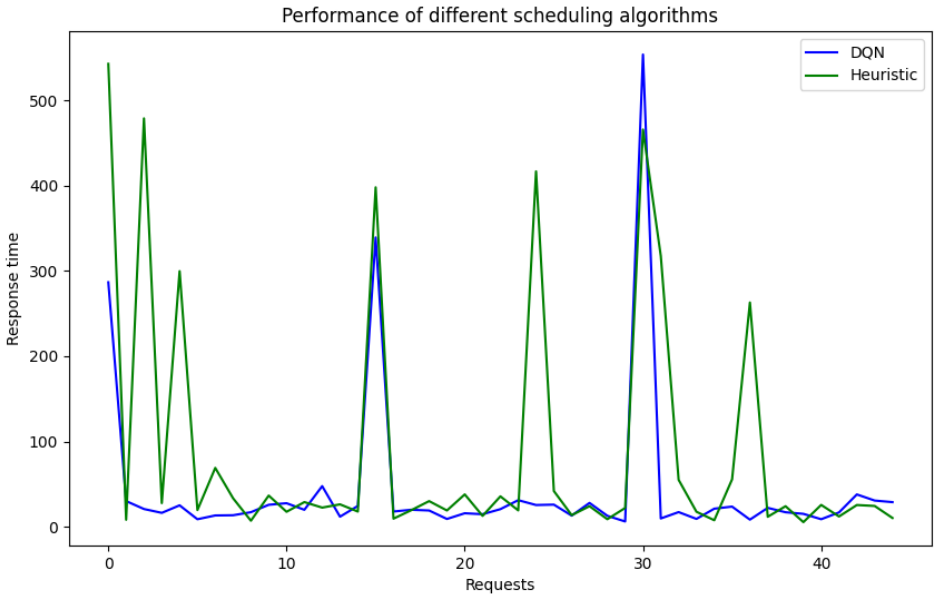

Figure 7.2: Function cold starts per instance using DQN and a heuristic.

evaluation. Across three batches of 15 function calls each, the DQN scheduler reduced cold starts to an absolute minimum of three, significantly outperforming the latency-greedy heuristic. The heuristic, due to its lack of consideration for cold starts, resulted in a total of 8 cold starts, approaching the absolute maximum of nine possible cold starts. The distribution of cold starts for the heuristic was uneven, with 3 cold starts in the first batch, 2 in the second, and 3 in the third. This uneven distribution highlights the heuristic's reactive nature and its inability to anticipate and avoid cold starts effectively. In contrast, the DQN scheduler's ability to learn and adapt to the environment's dynamics resulted in a more consistent and optimized scheduling strategy, minimizing cold starts and potentially improving overall system performance.

Table 7.4: Instance Settings.

Instance Type	**Availability**	**Latency (ms)**
Edge	100%	5-15
Private Cloud	100%	5-15
Public Cloud	100%	5-15

Comparison of Function Success Rates

This experiment aims to demonstrate the effectiveness of our DQN-based function scheduler in optimizing task allocation while considering the availability and failure

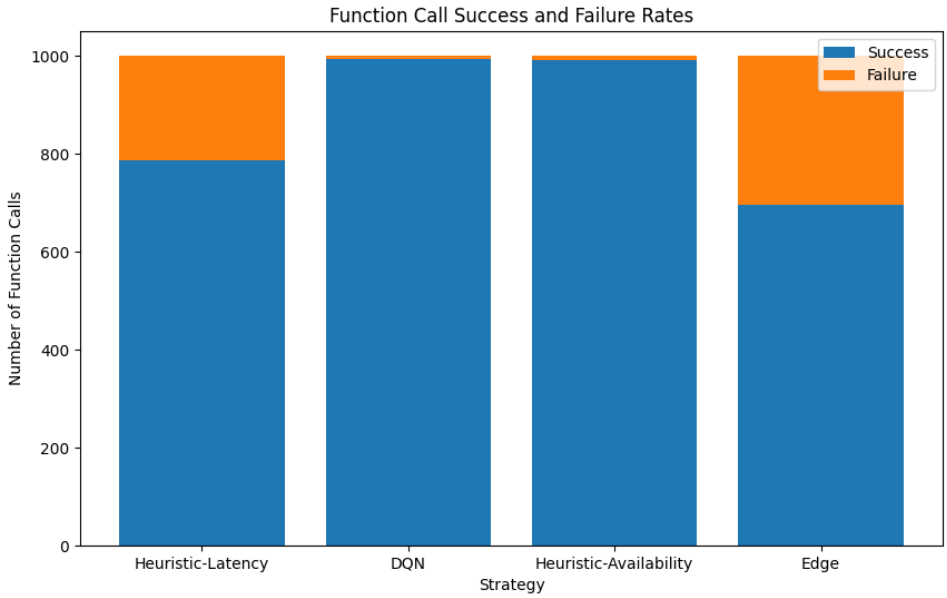

Figure 7.3: Failure rates per instance including an availability-greedy heuristic.

rates of serverless providers. We compare our DQN scheduler against three baseline heuristics: a latency-greedy heuristic, which prioritizes providers with the lowest latency without considering availability, an edge-only heuristic, which exclusively utilizes edge instances, and an availability-greedy heuristic, which considers the highest availability. The availability-greedy heuristic contains a modification: in case two instances share the same availability rate, then the best instance according to latency is chosen. By evaluating these strategies in a simulated environment with varying instance availability, we assess their impact on the success rate of 1000 function calls and the overall latency, with each function having a high probability of resulting in a cold start. Such scenarios may happen in the real world, especially over a longer period of time. Avoiding cold starts is particularly important in settings, where fast response times are needed while functions are infrequently dispatched. This analysis showcases the DQN scheduler's ability to learn and adapt to provider availability, maximizing successful function executions while maintaining low latency compared to the baseline heuristics.

Table 7.5: Instance Settings.

Instance Type	**Availability**	**Latency (ms)**
Edge	70%	5-15
Private Cloud	99%	16-25
Public Cloud	99%	25-40

Again, the results show that our DQN scheduler also outperforms the other baseline

strategies notably with a failure rate of only 0,7%. Specifically, Figure 7.3 shows the failure and success rates of each method, where our DQN scheduler reveals a significant reduction of function call failures out of 1000 function calls. Besides reducing failure rates, the DQN method also optimizes on latencies and cold starts, which results in an overall lower latency. It is important to note that cold starts occur frequently in our simulation environment. As expected from an availability of 70%, sending all functions to the edge results in an approximate failure rate of 30%. As the overall latency of the edge is the lowest, the latency-greedy heuristic considers scheduling to the edge in most cases, which results in a high failure rate too. Compared to the availability-greedy heuristic, both excel in avoiding failures, however, the DQN improves on response times (including cold start avoidance) compared to all other heuristics. While doing so, it can happen that the DQN scheduler compromises availability and risks a failure. As we see in Table 7.7, the edge instance has an already warm function container in contrast to the cloud instances. Despite having a low availability, the function gets scheduled to the edge anyway. This is due to the specific reward function used for training, where cold starts are higher penalized than failures. The latency greedy heuristic performed similar to the experiment before, favoring low latency, therefore potentially scheduling to cloud instances too, otherwise a failure rate of around 30% would be the case.

Table 7.6: Comparison of the latency-greedy heuristic, DQN, and edge-only scheduling.

	Mean Response Time (ms)	**Failure Rate (%)**
Heuristic-Latency	149.14	21.40
DQN	106.53	0.70
Heuristic-Availability	156.03	0.80
Edge	159.37	30.40

Table 7.7: Observation 1, results in action 1 (edge).

Instance	**Latency**	**Cold Start**	**Availability**
edge	1	0	70%
private-cloud	2	1	99%
public-cloud	11	1	99%

Comparative Analysis of Tabular Q-Learning and Deep Q-Networks

In our previous work [Sch22] we investigated the use of Q-learning, a tabular Reinforcement Learning approach, to solve the problem of function scheduling across different serverless providers. We trained the Q-table to excel in cold start avoidance where we could show similar performance to our experiment of Figure 7.2. Our investigation reveals that tabular Q-learning can be an effective method for addressing scheduling problems within the serverless computing domain, particularly in scenarios with limited state spaces. However, a fundamental limitation of the Q-learning approach lies in its susceptibility to the curse of dimensionality, also known as state space explosion. As the number

of state variables increases, the size of the Q-table grows exponentially, rendering it impractical for problems with large state spaces. For instance, considering just 10 binary state variables would result in a Q-table with 1024 entries. In our target environment, with over 20 such variables, a Q-table becomes entirely unmanageable. The inherent complexity of scheduling serverless functions, which raises the need for consideration of numerous parameters such as resource availability, cold starts, cost implications, and system information like CPU-load, highlights this limitation. Our proposed DQN-based scheduler addresses this challenge by leveraging the function approximation capabilities of deep neural networks, demonstrating the feasibility of handling high-dimensional state spaces inherent in complex serverless environments. Additionally, the Deep-Q-Networks are capable of handling a larger action state compared to the limited Q-table approach.

Assuming that we have a limited and discretized state representation for our Q-table, we model our evaluated state space, where

- Expected latency values range from 0-400 and are discretized into bins of 20.
- Cold starts are represented as booleans (0 or 1).
- Availability ranges from 0-100 and is discretized into bins of 10.
- CPU-load ranges from 0-100 and is also discretized into bins of 10.
- 3 FaaS providers are observed, each mapped to an action, resulting in 3 actions in total.

Then, we calculate the number of possible states for each provider (latency $\times$ cold start $\times$ availability $\times$ CPU load):

$$20 \times 2 \times 10 \times 10 = 4000$$

The combinatorial nature of the state space, with 4000 possible states per provider, leads to an exponential increase in complexity, reaching 4000^3 (64 billion) states when considering 3 providers. This results in $3 * 64$ *billion* $= 192$ *billion* state-action pairs. Assuming that 32-bit floats, which are equal to 4 bytes, are used to represent Q-values, this results in approximately 768 Gigabytes.

This huge memory requirement for this simplified state is not viable and shows the dimensions of the state-space explosion problem. Furthermore, this state representation is very coarse-grained where important information gets lost. Discretizing latency spaces into bins of 20 might not be sufficient for the serverless space. Additionally, discretizing CPU-loads like this also hides necessary details resulting in suboptimal scheduling decisions. So, a more fine-grained state representation is needed which would result in a much higher memory consumption. Especially, by adding a fourth provider or information about the specific function, such as adding function labels, would result in an exponential increase of memory consumption.

Unlike tabular Q-learning, which explicitly stores Q-values for each state-action pair, DQN employs a deep neural network to approximate the Q-function. This allows it to generalize across similar states, effectively handling large and even continuous state spaces without requiring explicit enumeration. This demonstrates that our DQN scheduler makes it possible to use a higher number of state parameters compared to tabular Q-learning, thus, making it feasible to observe various parameters at once

7.1.5 Load-Aware DQN Evaluation: Latency, Cold Start Calculation, Availability, CPU-Load

After comparing our basic DQN scheduler with simple baselines, we focus on its variant which considers load information about platform instances, assuming an environment where such information is available to the scheduler. Specifically, we want to evaluate if the CPU-load-aware DQN approach performs better than the cold-start-aware one. We expect both approaches to perform similar when it comes to avoiding cold starts, while still trying to minimize latencies. The CPU-aware DQN considers CPU loads of all instances in addition and tries to avoid high-loaded instances.

In the experiments we conduct in this section, we trained our DQN scheduler with an additional parameter per instance, namely CPU-load. We assume having access to white-box information such as CPU-load for this experiment. Indeed, there are ways to retrieve system information, such as built-in APIs or third party APIs serving this kind of information. For the sake of simplicity, we simulate CPU-load in order to focus on our scheduler's performance evaluation. Generally, there is always a possibility of writing custom services and deploy them on the instances for observation. A common practice is to use monitoring tools like Prometheus.[3] It records system information in a time series database, which can then be queried. It works for obtaining system information of cloud deployments too. When deployed on Kubernetes, metrics like CPU-load can be obtained by leveraging Kubernetes' monitoring capabilities.

The setup is similar to the previous DQN algorithm. We trained it with less iterations, 800 in total, as it converges quite fast for the specific setup we use for our experiments. The batch size is equal to 32, but we set the steps per episode up to 80 in order to capture CPU-load patterns better. We also use the same weighted averaging method to represent latency information.

$$\overline{L_i} = 0.1l_i^{(3)} + 0.3l_i^{(2)} + 0.6l_i^{(1)}$$

For the reward function, we decided on a similarly simple one which penalizes high CPU-loads in addition, as we define in Chapter 5.

Analysis of CPU-load-aware DQN scheduler performance

In this experiment, we evaluate how scheduling decisions are made given specific situations. As an additional parameter was introduced, the scheduler now has to make trade-offs,

[3] `https://prometheus.io/`

Table 7.8: Summary of DQN-parameters.

Steps per Episode	**Training Iterations**	**Batch Size**
80	800	32
Replay Buffer α	**Replay Buffer** β	**Replay Buffer Capacity**
0.5	0.5	60 000
Learning Rate α	**Discount Factor** γ	
0.001	0.99	

whether to go for a cold start, to overload an instance or to risk a failure. According to the reward function, the scheduler should rather avoid overloaded instances (CPU-load above 80%) over cold starts or failures.

Figure 7.4 shows how functions are scheduled out of a total 25 functions. The instance in the private cloud starts with high load, thus, the scheduler is expected to avoid that instance. Moreover, we assume that function calls are scheduled within a few milliseconds before the first function returns which leads to a continuous increase of CPU-load. By doing so, decisions based on CPU-load can effectively be shown.

The results are interesting since the algorithm now has to choose whether to avoid cold starts or high loads in situations where both cannot be prevented. In most cases the function call was placed on less loaded instances such as the edge or the public-cloud, where the edge was the preferred choice generally due to its low latency.

The following observations represent the states and the chosen actions which map to the instances respectively (1 to edge, 2 to private-cloud, 3 to public-cloud). The observation space represents 4 parameters per each of the 3 instances (latency measure, cold start, availability, CPU-load). We normalize values to improve the training and decision performance, which is why latencies were assigned low integer values. Precisely, we divide latencies through 10 and floor the result, thus, discretizing the values into baskets of size 10.

The observations make it clear how the agent reacts to different situations. Its priority lies in avoiding cold starts and loaded instances. In Observation 3 (Table 7.11) the agent decides on the last instance (public cloud) despite having a higher latency compared to the second (private cloud). The first instance (edge) is avoided due to the cold start measure. Observation 1 (Table 7.9) is a difficult case. All instances result in a cold start, thus, the penalty for all instances would be high. The probable explanation is a lack of diverse training data which was generated by the simulations and the algorithm was not able to optimize its policy in specific states like this. In this scenario with a low number of function calls, the scheduler was able to effectively avoid cold starts except in observation 1. Due to the high availability, no failures occured. The mean response time

resulted in 45.1 ms and the failure rare in 0%.

- Mean Response Time (DQN): 45.41 ms
- Failure Rate (DQN): 0.00

In real scenarios, a more sophisticated reward calculation might be beneficial as it might deliver better results in a wider range of cases, but in our experiments this suffices. We want to show the interconnection of state parameters with a slight focus on avoiding high CPU-loads as penalties reach up to -60. We still gradually increase penalties in our reward function to indicate that a higher load results in a worse result. However, these results are very promising as a refined model with a more sophisticated reward calculation, and more thorough training, would recognize invocation and instance patterns, therefore being able to efficiently decide on the best instance.

Table 7.9: Observation 1, resulted in action: 2.

Instance	Latency	Cold Start	Availability	CPU Load
edge	1	1	99%	0%
private-cloud	2	1	99%	90%
public-cloud	3	1	99%	10%

Table 7.10: Observation 2, resulted in action: 1.

Instance	Latency	Cold Start	Availability	CPU Load
edge	1	0	99%	20%
private-cloud	2	0	99%	100%
public-cloud	3	0	99%	10%

Table 7.11: Observation 3, resulted in action: 3.

Instance	Latency	Cold Start	Availability	CPU Load
edge	2	1	99%	30%
private-cloud	2	0	99%	100%
public-cloud	4	0	99%	10%

7.2 End-To-End System Evaluation

7.2.1 A real-world use case

This research was conducted in collaboration with the Austrian Institute of Technology (AIT) and its subsidiary, Leichtmetallkompetenzzentrum Ranshofen (LKR),[4] which

[4] https://www.ait.ac.at/ueber-das-ait/center/center-for-transport-technologies/lkr-leichtmetallkompetenzzentrum-ranshofen

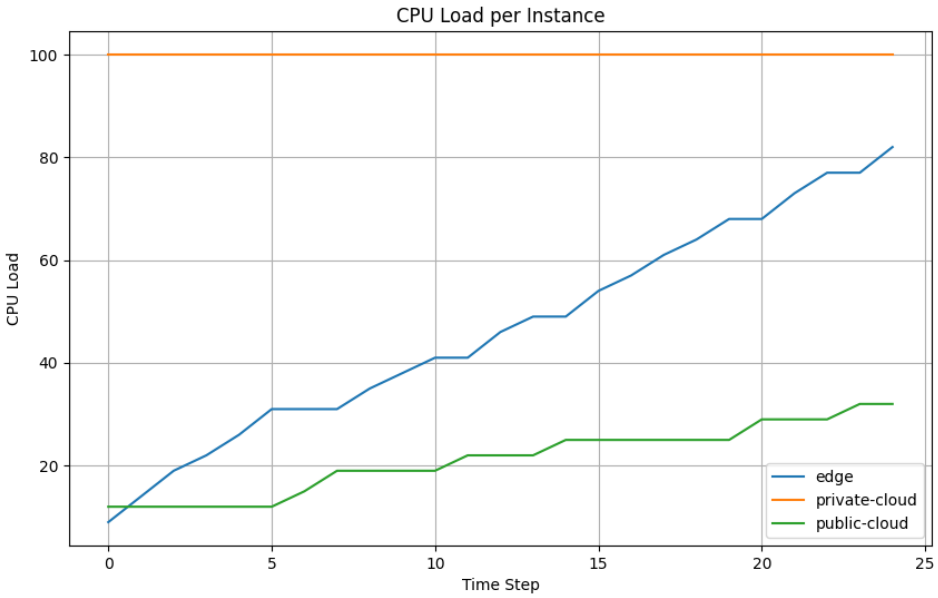

Figure 7.4: Evolution of CPU load for each instance.

provided a real-world use case for the system's application. LKR is specialized in light metal manufacturing and develops (also in cooperation with other companies) new manufacturing techniques to make production more efficient.

Currently, they are upgrading their IT infrastructure by digitalizing their manufacturing processes on the software side to take a step forward to Industry 4.0 standards. From this, they expect better efficiency in general by collecting various data, such from those produced by machine sensors.

In parallel, they are developing a proprietary anomaly detection model for one of their processes, light metal casting. We consider running this model within a serverless function as highly beneficial, as specific logic bound to a use case is decoupled from the remaining architecture. This separation enhances maintainability, simplifies updates, and provides greater control over code. Furthermore, the collaboration with AIT and LKR ensures the system's practical relevance and applicability to real-world industrial settings.

The integration of real-time data collection and processing, coupled with the anomaly detection model, enables LKR to significantly enhance the efficiency of their manufacturing processes. By continuously monitoring sensor readings, the system facilitates automated decision-making, enabling autonomous adjustments to the manufacturing environment, such as regulating temperature. Moreover, in situations where environmental adjustments are insufficient to address an anomaly, the system is capable of halting the entire process, thereby preventing the further consumption of resources such as electricity and material and mitigating potential production losses. This capability to dynamically respond to real-

time conditions, ranging from fine-grained adjustments to complete process shutdowns, is a key advantage of the implemented system, ultimately contributing to optimized resource utilization and improved product quality.

A key requirement for this system is the ability to produce results in real-time. The upper limit of producing a result is 5 seconds to support manufacturing processes. Therefore, evaluating the performance of the whole pipeline of services is critical, i.e. the time the system needs to collect sensor data in the gateway, passing this further to the context broker, which serves this data to applications subscribing to and processing this data.

In this use case this application runs our scheduler, calls serverless functions and processes the results. For the serverless platform, we decided on Nuclio,[5] which excels in performance needed in high performance settings such as manufacturing processes where a high volume of function invocations is expected. The overall experimental architecture is depicted in Figure 7.5.

7.2.2 Testbed Setup

To evaluate the performance of our proposed system, we constructed an end-to-end testbed on a single machine with the hardware described in Table 7.12. This setup, while providing a controlled environment for experimentation, aims to emulate a realistic IIoT edge deployment. The testbed comprises several key components: **(1) Simulated OPC UA Server**: This simulates the machine sensor data source, generating data streams that mimic real-world sensor readings with varying patterns with a fixed frequency. **(2) OPC UA Gateway**: This component collects the simulated sensor data and forwards it to the context broker. **(3) NGSI-LD Context Broker**: Acting as the central information hub, the context broker stores and manages the sensor data, making it accessible to other applications. **(4) Application and Scheduler**: This microservice continuously monitors the context broker for new sensor data by subscribing to it and employs a DQN-based algorithm to decide which Nuclio instance is best suited to process the incoming data. Both run within the same container in this experiment. However, whether running the scheduler outside or coupled as part of the application is up to the user. **(5) Nuclio Function Instances**: These serverless function instances host the anomaly detection model. The scheduler dynamically assigns incoming data to these instances for processing. **(6) Logging**: To capture performance metrics, we integrated logging mechanisms to record timings, such as data ingestion latency, scheduling decisions, and function execution times. Basically, we create log files for each container logging timestamps at relevant locations in code. This comprehensive setup allows us to systematically analyze the behavior and performance of the system under different conditions and workloads.

We run the system using Docker Compose which is widely used for local testing of multi-container applications. We run the whole end-to-end experiment for roughly 15 minutes where the system collects and processes data of four sensor nodes in total. Our goal is to test the feasibility of such a system in terms of stability and performance.

[5] `https://nuclio.io/`

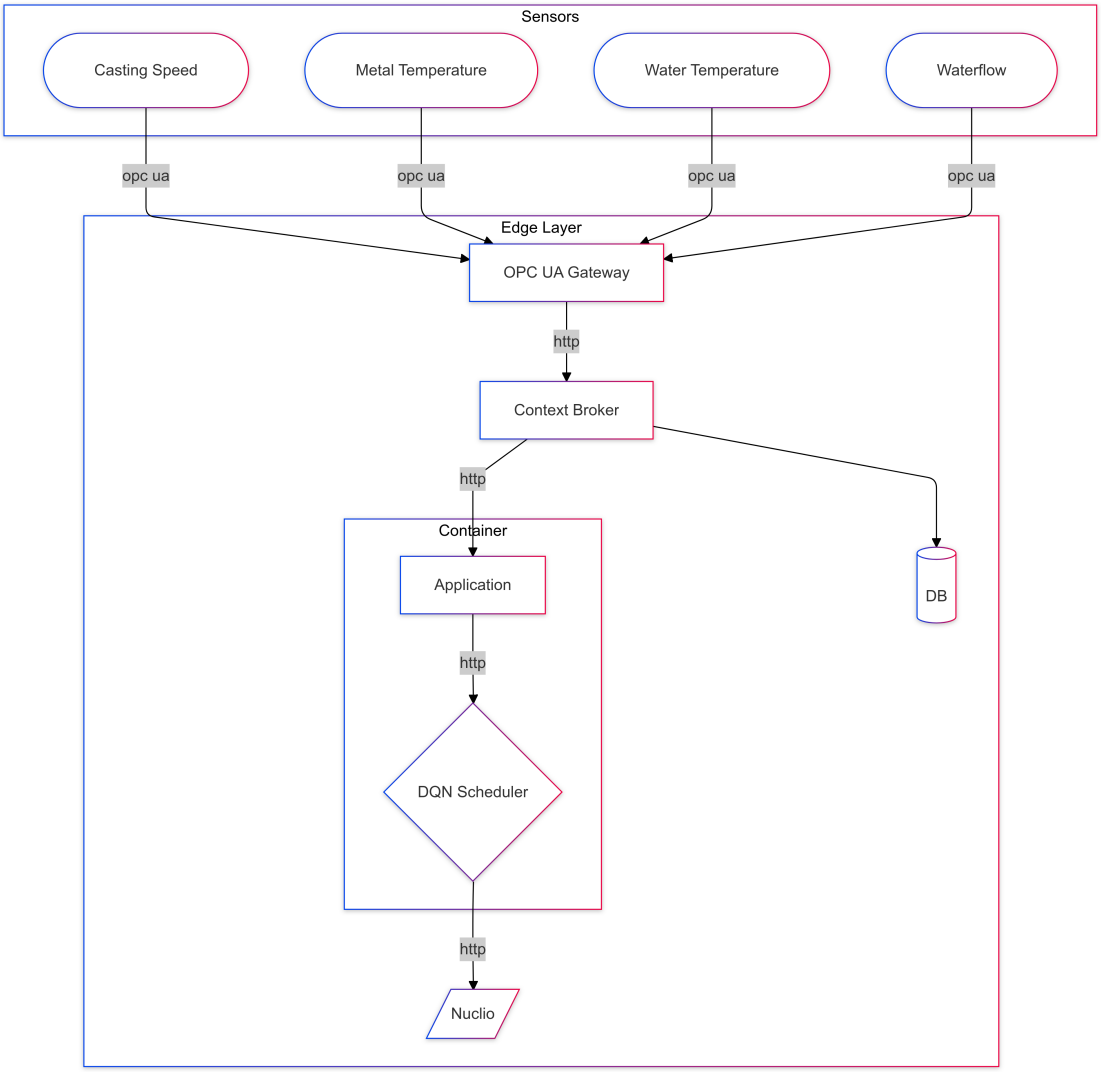

Figure 7.5: Architecture used for this experiment. The DQN schedular is bundled in the same container with the application for simplicity.

Instead of sending each single value over the network, we preprocess the values in the gateway and send them in batches of 10. This coincides with the requirements of the anomaly detection model, which receives as input and processes only 10 float values at a time. As our focus here is more on integration aspects and feasibility testing, rather than evaluating the decision quality of our scheduler, we only deploy a single serverless instance, which is responsible for all function calls. Note that the scheduler is still operating on the function serving path, thus contributing a small amount of delay ($\sim$2 milliseconds) to the perceived response time, that is mainly attributable to invoking it over its API endpoint and executing the DQN.

Table 7.12: Hardware details of the testbed host.

Experiment Environment	
Operating System	Microsoft Windows 11 Home 64-Bit
CPUs	Intel Core i7-10875H @ 2.30GHz, 2304 MHz, 8 cores
Main Memory	16 GB DDR4 RAM
GPU	Nvidia GeForce RTX 2080 Super MAX-Q | 16 GB GDDR5 RAM

Simulated OPC UA Server

This component realistically simulates possible sensor values for our specific use case. Specifically, the following sensor readings are captured for this experiment, where all values have the *float* data type.

- Casting speed, simulated using a distribution centered around a mean value of 2.
- Metal temperature, using a distribution of the mean value 923.15.
- Water temperature, using a distribution of the mean value 294.15.
- Water flow, using a distribution of the mean value 9.

Furthermore, we captured real sensor frequencies from manufacturing processes on the factory side. For this, we connected onto a VM running on the edge server, subscribed to real machine sensors and logged frequencies and values into a file. In the process of light metal casting, each sensor provides 20 values per second, which means that every 50 milliseconds one value is produced and published by the OPC UA server per sensor node.

Logging and Measurements

Our key performance metric is end-to-end latency. This metric is an expression of the *sensing-to-actuation delay*, which is critical for the use case under study. In particular, in the anomaly detection scenario that we experiment with, the latency components in the end-to-end path are the following: (i) transporting data from sensors to the application ingestion endpoint via the gateway and the context broker; (ii) carrying out preprocessing application logic, i.e., preparing the serverless function invocation and transmitting it to the scheduler; (iii) scheduling the function to the serverless platform; (iv) executing the function and receiving the result by the respective application component. Notably, this could be another serverless function as part of a serverless workflow orchestrated by the application, that would be responsible for actuation upon anomaly detection – we do not implement such logic in this experiment.

7.2.3 Results

This section presents the findings of the end-to-end experiment designed to evaluate the performance and stability of our serverless approach in an IIoT environment in an end-to-end style. The experiment involved running the system under a consistent workload for 15 minutes, capturing detailed logs from each component (gateway, scheduler, and Nuclio functions), and monitoring CPU load. The analysis focuses on key performance indicators, including individual component durations, overall round-trip latency, and the distribution of these durations to identify potential bottlenecks and assess system behavior under load. Our aim is to show feasibility in terms of the set real-time requirement of five seconds as a maximum, which this system meets. The results are presented through a series of visualizations and statistical analyses to provide a comprehensive understanding of the system's performance characteristics.

We could identify rough bottlenecks in our system but we can not name specific problems as this would require thorough testing in a real deployment. However, we could observe a good level of stability, especially for the scheduler. In addition, the minimum requirement of an overall processing time, scheduling and processing of a maximum of 5 seconds was met even though of the presence of outliers pushing the end-to-end latency up to 1.5-2 seconds, which is extractable from following metrics.

- Sensor value handling at the gateway resulted in a mean of 111.66, 118.15, 111.10 and 124.97 ms respectively per sensor.
- The schedulers mean processing time for scheduling and calling a function was 106.41 ms.
- The total end-to-end latency mean value results in 222.90 ms.

Sensor and OPC UA Gateway Analysis

Here we show how the gateway handled the four sensors sending data concurrently. Figure 7.6 reveals crucial insights into the performance characteristics of the sensor data ingestion process within the tested system. Each line in the plot represents the time taken to process a batch of sensor readings from a different sensor. A prominent pattern observed across all sensors is the recurring spikes in duration, where processing times abruptly jump from a baseline of under 100 milliseconds to 400-500 milliseconds. This recurring pattern suggests a periodic bottleneck or delay affecting all sensor data streams. But in general, they range around 100-120 ms.

These spikes in sensor data processing times have significant implications for the overall system performance. In real-time IIoT applications, where timely data processing is crucial, such unpredictable delays can hinder the system's ability to respond promptly to events and anomalies. The variability in sensor data ingestion times introduces jitter and inconsistency in the data pipeline, potentially affecting the accuracy and effectiveness of

applications relying on these data. However, these spikes do not disturb efficiency of our system in our specific LKR use case as the threshold of 5 seconds provides space.

Several factors could contribute to these spikes such as network latency resulting for inconsistency, specific implementations in the gateway or processing in the gateway. Another reasonable possibility lies in our testbed which consists of a resource-limited host in terms of memory and processing power or implementation choices for the gateway itself. To identify the reasons behind the spikes would raise the need to rerun experiments on real deployments on the target servers. So, further analysis on an enhanced deployment with increased compute capacity and capabilities such as network monitoring, profiling of the gateway code, and experimentation with different buffering strategies, can help pinpoint the source of the delays and guide optimization efforts. By addressing these performance bottlenecks in the sensor data ingestion pipeline, the overall system's responsiveness and reliability can be significantly improved.

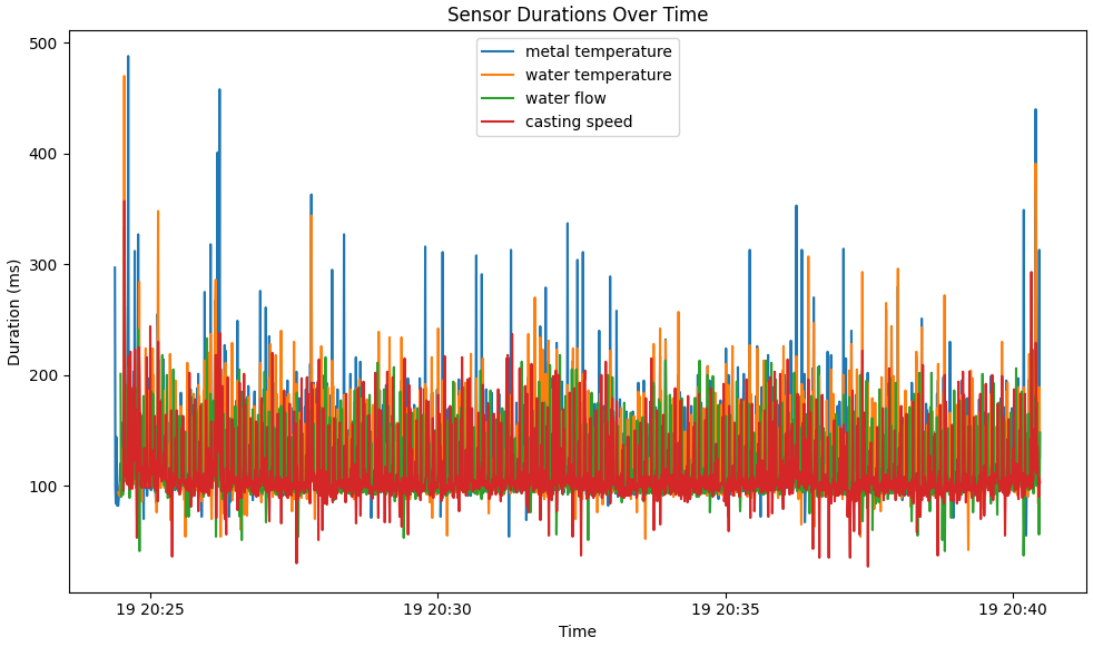

Figure 7.6: Durations for each sensor over 15 minutes.

Overall end-to-end latency

Figure 7.7 visualizes the variability and consistency of the processing times within the tested system. It illustrates how frequently different processing durations occur for each component, i.e. scheduler and sensors, and the overall end-to-end latency. By examining the shape and spread of these distributions, we can gain insights into potential bottlenecks and the overall predictability of the system's performance.

The blue bars, representing the sensor data ingestion durations, reveal a distinct bimodal distribution. This indicates that the sensor processing times fall into two primary categories: a cluster of very short durations (close to 0 milliseconds) and a smaller cluster of longer durations around 250 milliseconds. This bimodal pattern aligns with the observations from Figure 7.6, confirming the presence of frequent, periodic spikes in sensor processing times. These spikes introduce significant variability in the sensor data pipeline, potentially affecting the responsiveness of the system to real-time events.

In contrast, the orange bars, representing the scheduler's processing durations, show a more even distribution. This suggests that the scheduler's performance is relatively consistent together with the serverless Nuclio function running the anomaly detection model, with no single processing duration being overly dominant. Although there is a small deviation of up to 300-400 ms, it can still be considered as consistent. This is desirable in a scheduling component, as it ensures more predictable behavior and avoids introducing additional jitter into the system.

The green bars depict the total end-to-end latency, where characteristics of the scheduler and sensors are combined. The distribution is skewed towards shorter durations, primarily due to the frequent fast processing times of the sensor. However, the noticeable tail of longer durations highlights the influence of both the sensor's spikes and the scheduler's inherent variability.

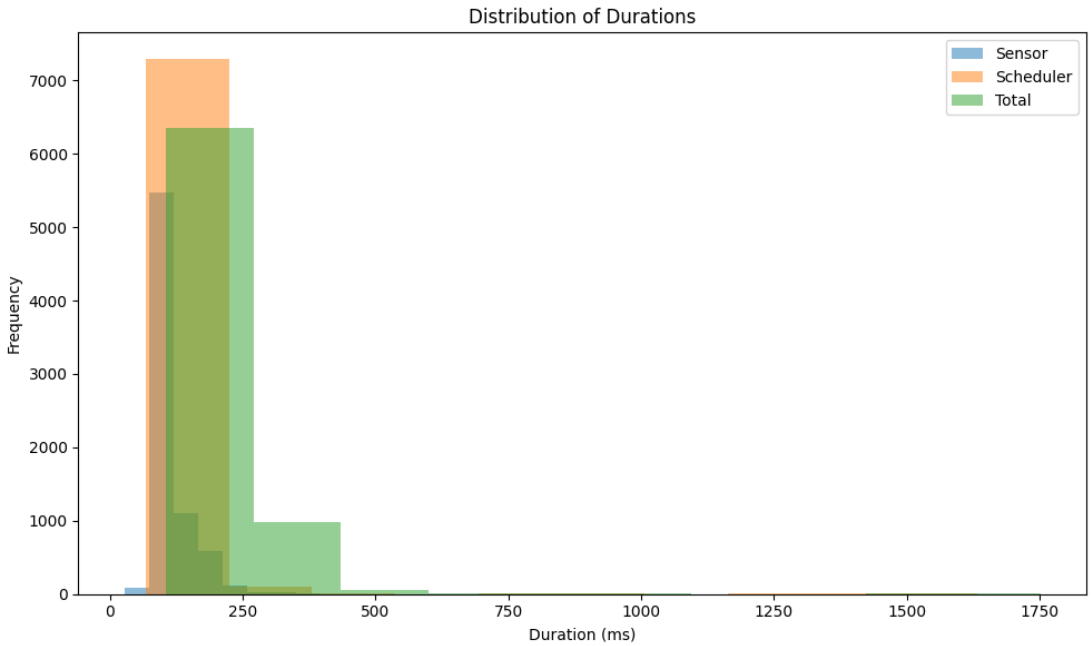

Figure 7.7: Represents the distribution of all durations measured.

Closer Analysis of Individual Component Performance

Figures 7.8 and 7.9 provide valuable insights into the relationship between individual component performance and the overall responsiveness of the system. These plots help identify which components have the most significant impact on the total end-to-end latency, which is a critical measure of the system's ability to handle real-time data processing.

In particular, Figure 7.8 plots the end-to-end latency with respect to the latency introduced by the process of ingesting sensor data, i.e., collecting data from the sensors (OPC UA servers), pushing them to the context broker, and eventually delivering them to the application. The figure shows a clear positive correlation. This means that as the sensor data ingestion time increases, the overall end-to-end latency also increases proportionally. This observation confirms that the sensor data processing stage including the gateway is a major contributor to the overall latency of the system. Any delays or inefficiencies in handling sensor data directly translate to increased end-to-end latencies, but with regard to our requirement, this is still acceptable.

Conversely, end-to-end latency exhibits a weaker correlation with the delay induced by the execution of the scheduler, as Figure 7.9 shows. While longer scheduler processing times do contribute to increased end-to-end latencies, the impact is less pronounced compared to the sensor processing in the gateway. This suggests that the scheduler, while still a factor in overall performance, is not the primary bottleneck. There might be opportunities for minor optimizations in the scheduler, but the focus should primarily be on further analyses in real-world deployments on stronger and more stable compute infrastructure, and addressing the more significant impact of the sensor data ingestion process if it still holds.

These findings highlight the importance of optimizing the sensor data pipeline to minimize latency and improve overall system responsiveness. Strategies such as reducing network latency, optimizing implementation of the gateway, and streamlining data processing within the gateway can significantly reduce sensor processing times and, consequently, the overall end-to-end latency. By prioritizing these optimizations, the system can achieve better real-time performance and ensure timely processing of critical sensor data in real-time critical applications where latency requirements are more stringent.

That said, while the observed sensor delays might raise concerns about real-time performance, it is crucial to consider them within the context of our application's requirements. The system's design specifies a maximum allowable sensor-to-actuation latency of 5 seconds for our light metal casting use case. Even with the occasional spikes in sensor processing time, the vast majority of end-to-end latencies remain well below this threshold. Therefore, despite the variability observed in the sensor data ingestion process, the system meets the performance requirements for this specific use case. This highlights the importance of aligning performance evaluation with the practical needs and tolerances of the target application.

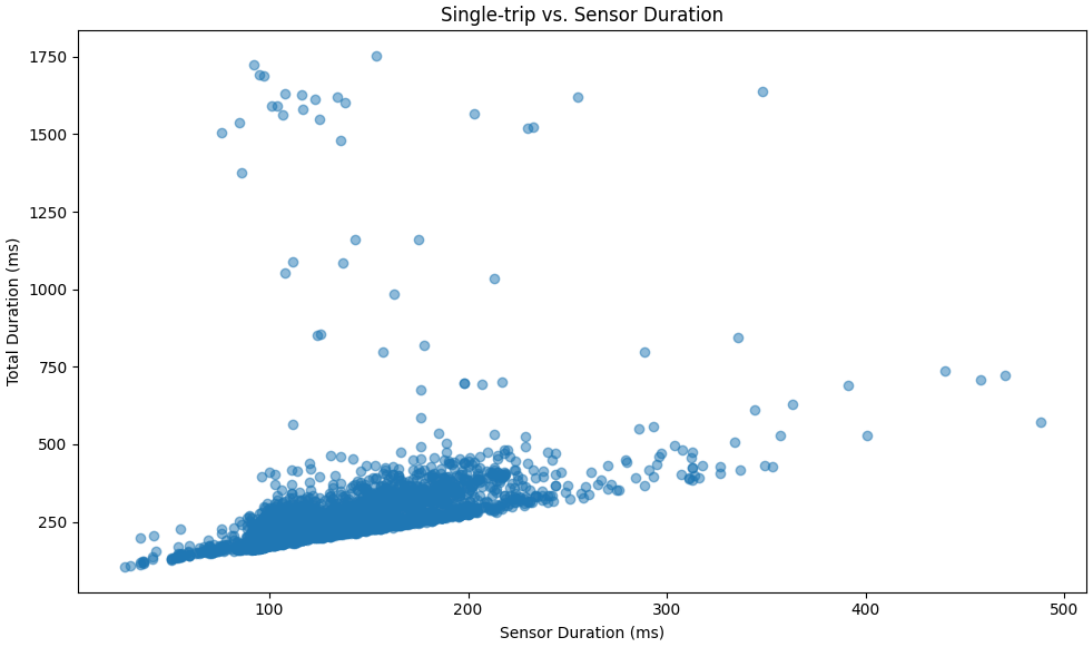

Figure 7.8: Scatter plot representing influence of the gateway and sensors on overall performance.

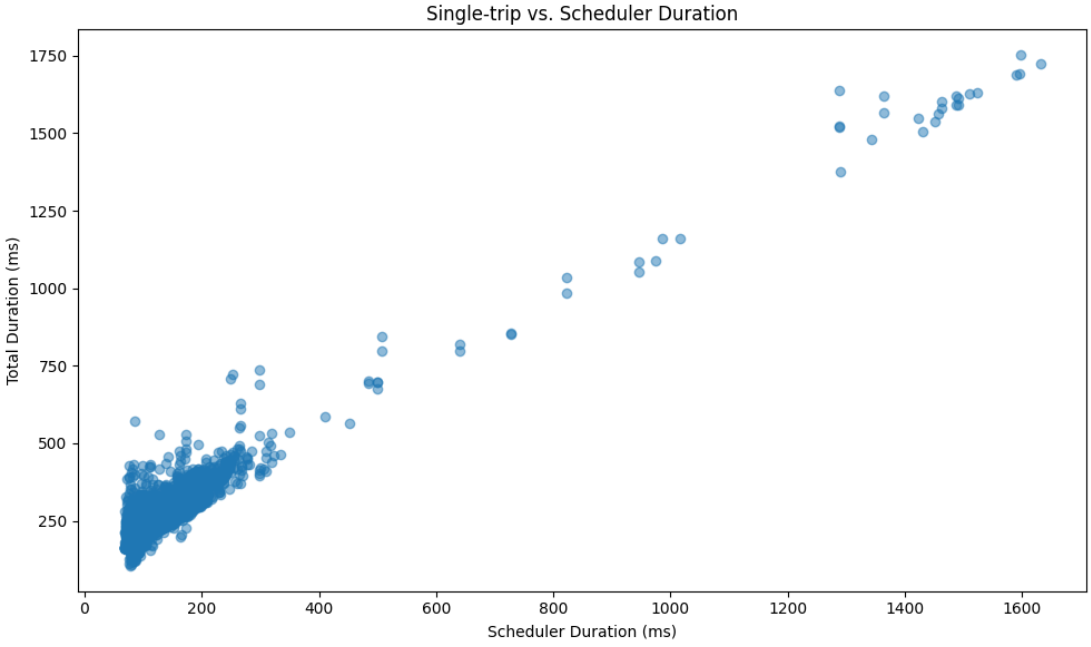

Figure 7.9: Scatter plot representing influence of the scheduler on overall performance.

CHAPTER 8

Conclusion

This chapter revisits the posed research questions and highlights the major contributions and findings of this thesis towards answering them. This work addressed a broad spectrum of issues that pertain to serverless-supported industrial processes, from architecture specification and interoperability support, to algorithmic aspects of serving IIoT application workloads. Naturally, and given its scope, this work also comes with some limitations. These give rise to interesting directions for future study, with which we conclude this thesis.

8.1 Contributions

RQ1: How can a serverless approach be applied in an industrial IoT edge setting, what benefits can it bring about, and how can existing standards for data collection and representation be leveraged to improve on integration capabilities and interoperability?

We could answer this question by providing an implementation of such a system which addresses the challenges of incorporating serverless and data collection. By providing an architecture, which is able to ingest data from sensors by having a uniform interface and supports widely used standards such as OPC UA, and processes data concurrently, this system offers the possibility of integrating numerous types of different sensors relevant in manufacturing and other industrial environments. By applying the NGSI-LD standard, the integrated Orion Context Broker improves interoperability significantly by adding relevant context information to data. It also serves as a single endpoint allowing different services using data as from sensors.

RQ2: How to achieve high-performance function serving in Industrial IoT in order to meet requirements for real-time data processing?

Answering this research question involved a deep dive into the nature of serverless functions and what challenges arise when employing this model into a system. We came to the conclusion that serverless raises difficult challenges when it comes to executing functions with specific requirements, such as response time, success rate or balancing function invocation across different providers based on CPU-load. That said, we discuss the hybrid use of cloud and edge resources, as additional servers ensure stability of such an industrial IoT system. On-premise edge servers might get overcharged as they are subjected to unpredictable fluctuations in workload, particularly during peak processing times or unexpected events. Cloud resources, with their elastic scaling properties, can absorb these spikes, ensuring continuous operation and preventing service disruptions. Therefore, we introduce an approach which uses DQNs to automate the decision on which serverless provider suits best the given requirements on a per function invocation basis. For example, in our discussed use case of anomaly detection in industrial processes, the anomaly detection model is designed to run as a serverless function and has strict performance requirements. Thus, the DQN scheduler makes decisions based on the lowest response time. As cold starts are included into response times, our scheduler excels at avoiding them, thus, maximizing response-time based performance. We should note that we introduce different variants of our DQN-based scheduler and discuss further possible extensions, to account for different assumptions regarding their operating environment.

RQ3: How does such a system affect performance in IIoT systems where performance is a critical key consideration?

Such as system needs to be feasible, which is why we also run end-to-end experiments by deploying the whole system and test it. Although, we did not deploy it in a real factory, but instead the system was run on a local testbed, the fidelity of this experiment is high. In corporation with AIT, we were able to collect data like request patterns from real processes by using historical data and, in addition, connecting to sensor readings during a real experiment and log all relevant results. We came to the conclusion that such a system works perfectly fine in overall without significant bottlenecks, but with recommendations of improvements, as we identified some minor issues at the data ingestion level, i.e. the gateway. This system effectively facilitates the transition to Industry 4.0 with extensibility in mind. Deploying this whole system together with the anomaly detection model provided by AIT for our use case can result in significant savings in terms of material, resource use and costs, because it allows real-time evaluation of the process. In the worst case for example, if an inveterate error occurs, this whole system is able to identify this error and stop the whole process instead of wasting further material and resources on a wasted product.

Importantly, we found that our first cold-start aware DQN algorithm improves on cold starts by a factor of roughly 3 according to our evaluated scenario. When it comes to minimizing failures, the DQN scheduler achieved up to 8.32 times less failures while still improving on response times by mainly avoiding cold starts by a factor of up to 1.25.

8.2 Limitations and Future Work

As our system was not tested in a factory environment, this could be an aspect for future work, namely to measure how this system performs in a real deployment. Compared to a local testbed, problems can be identified which are otherwise hidden in a local deployment. This way a better measure in terms of savings could be achieved by tracking how much electricity, time or material was saved.

A limitation of the DQN algorithm is its fixed action space which allows to schedule only to a predefined set of providers. Interesting future work would be to find a way to mitigate or solve this issue. Evaluating the use of other algorithms like PPO for the problem of function scheduling could be valuable, as PPO is able to handle continuous action spaces.

The question of running a single scheduler for all applications or running multiple schedulers with different policies remains open. A scheduler per application would involve custom requirements and priorities which a single scheduler could not easily handle. We explored the option of assigning functions different labels reflecting their priorities by weighting the reward function. However, our early experimental investigation of this approach revealed that it makes training more complex. Implementing in our system and evaluating a DQN scheduler that addresses these challenges, and efficiently and flexibly balances the competing demands of different coexisting functions is an important subject for future study.

Lastly, interesting directions for future work are related with different parameters to consider into our state and action space, and to take into account additional factors, particularly related with the costs to offload workloads to external cloud providers. This would influence the design of appropriate reward functions. Costs matter also for on-premise edge instances as resources are consumed per each function call. Therefore, considering the latter jointly with the different pricing strategies of serverless providers, as well as with the respective service-level agreements offered and the attainable performance is a problem worth studying.

Overview of Generative AI Tools Used

In a limited number of cases, Google Gemini Pro 1.5 (`https://gemini.google.com/app`) was used. It served as an aid for writing in the English language. It was used sparingly for improving language only based on self written paragraphs.

Übersicht verwendeter Hilfsmittel

Für diese Diplomarbeit wurde Google Gemini Pro 1.5 (`https://gemini.google.com/app`) als Hilfestellung verwendet. Dieses Tool wurde nur für die Verbesserung der englischen Sprache verwendet.

List of Figures

List of Tables

Bibliography

[AGS+21] Susan Amin, Maziar Gomrokchi, Harsh Satija, Herke van Hoof, and Doina Precup. A survey of exploration methods in reinforcement learning. *CoRR*, abs/2109.00157, 2021.

[AJJ+24] Auday Al-Dulaimy, Matthijs Jansen, Bjarne Johansson, Animesh Trivedi, Alexandru Iosup, Mohammad Ashjaei, Antonino Galletta, Dragi Kimovski, Radu Prodan, Konstantinos Tserpes, George Kousiouris, Chris Giannakos, Ivona Brandic, Nawfal Ali, André B. Bondi, and Alessandro V. Papadopoulos. The computing continuum: From iot to the cloud. *Internet Things*, 27:101272, 2024.

[ARB21] Siddharth Agarwal, Maria A. Rodriguez, and Rajkumar Buyya. A reinforcement learning approach to reduce serverless function cold start frequency. In *2021 IEEE/ACM 21st International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, pages 797–803, 2021.

[BBFL19] Duncan Bees, Martin Bauer, Mike Fisher, and Wenbin Li. NGSI-LD API: for Context Information Management. White Paper 31, ETSI, January 2019.

[BDJ+22] Philipp Brauner, Manuela Dalibor, Matthias Jarke, Ike Kunze, István Koren, Gerhard Lakemeyer, Martin Liebenberg, Judith Michael, Jan Pennekamp, Christoph Quix, Bernhard Rumpe, Wil M. P. van der Aalst, Klaus Wehrle, Andreas Wortmann, and Martina Ziefle. A computer science perspective on digital transformation in production. *ACM Trans. Internet Things*, 3(2):15:1–15:32, 2022.

[BKJS21] Ataollah Fatahi Baarzi, George Kesidis, Carlee Joe-Wong, and Mohammad Shahrad. On merits and viability of multi-cloud serverless. In *Proc. ACM Symposium on Cloud Computing (SoCC)*, 2021.

[BPS+22] Lorenzo Bacchiani, Giuseppe De Palma, Luca Sciullo, Mario Bravetti, Marco Di Felice, Maurizio Gabbrielli, Gianluigi Zavattaro, and Roberto Della Penna. Low-latency anomaly detection on the edge-cloud continuum for industry 4.0 applications: the SEAWALL case study. *IEEE Internet Things Mag.*, 5(3):32–37, 2022.

[BQT23] Luciano Baresi, Giovanni Quattrocchi, and Inacio Gaspar Ticongolo. Dependency-aware resource allocation for serverless functions at the edge. In *Proc. 21st International Conference on Service-Oriented Computing (ICSOC)*, 2023.

[Bus24] Axel Busboom. Automated generation of OPC UA information models - A review and outlook. *J. Ind. Inf. Integr.*, 39:100602, 2024.

[CSB+19] Flavio Cirillo, Gürkan Solmaz, Everton Luís Berz, Martin Bauer, Bin Cheng, and Ernö Kovacs. A standard-based open source iot platform: FIWARE. *IEEE Internet Things Mag.*, 2(3):12–18, 2019.

[Cui18] Yan Cui. I'm afraid you're thinking about aws lambda cold starts all wrong. `https://hackernoon.com/im-afraid-you-re-thinking-about-aws-lambda-cold-starts-all-wrong-` January 15th, 2018. Accessed: 09.01.2025.

[DCD23] Schahram Dustdar, Victor Casamayor-Pujol, and Praveen Kumar Donta. On distributed computing continuum systems. *IEEE Trans. Knowl. Data Eng.*, 35(4):4092–4105, 2023.

[DIKK18] Jürgen Dobaj, Johannes Iber, Michael Krisper, and Christian Kreiner. A Microservice Architecture for the Industrial Internet-Of-Things. In *Proc. 23rd European Conference on Pattern Languages of Programs (EuroPLoP)*. ACM, 2018.

[ETS22] ETSI GS MEC 003. *Multi-access Edge Computing (MEC); Framework and Reference Architecture*, March 2022. V3.1.1.

[FR24] Mauro Femminella and Gianluca Reali. Comparison of reinforcement learning algorithms for edge computing applications deployed by serverless technologies. *Algorithms*, 17(8), 2024.

[FSP+24] Daniel Fireman, Paulo Silva, Thiago Emmanuel Pereira, Luis Mafra, and Dalton Valadares. Prebaking runtime environments to improve the faas cold start latency. *Future Generation Computer Systems*, 155:287–299, 2024.

[GFD22] Philipp Gackstatter, Pantelis A. Frangoudis, and Schahram Dustdar. Pushing serverless to the edge with webassembly runtimes. In *Proc. 22nd IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGrid)*, 2022.

[GGW+24] Muhammed Golec, Sukhpal Singh Gill, Huaming Wu, Talat Cemre Can, Mustafa Golec, Oktay Cetinkaya, Felix Cuadrado, Ajith Kumar Parlikad, and Steve Uhlig. MASTER: Machine Learning-based Cold Start Latency Prediction Framework in Serverless Edge Computing Environments for Industry 4.0. *IEEE Journal of Selected Areas in Sensors*, 2024. In press.

[GHIU17] Markus Graube, Stephan Hensel, Chris Paul Iatrou, and Leon Urbas. Information models in OPC UA and their advantages and disadvantages. In *Proc. 22nd IEEE International Conference on Emerging Technologies and Factory Automation, ETFA*, 2017.

[HMvH+17] Matteo Hessel, Joseph Modayil, Hado van Hasselt, Tom Schaul, Georg Ostrovski, Will Dabney, Dan Horgan, Bilal Piot, Mohammad Azar, and David Silver. Rainbow: Combining improvements in deep reinforcement learning, 2017.

[HS24] Razin Farhan Hussain and Mohsen Amini Salehi. Resource allocation of industry 4.0 micro-service applications across serverless fog federation. *Future Generation Computer Systems*, 154:479–490, 2024.

[JSSS+19] Eric Jonas, Johann Schleier-Smith, Vikram Sreekanti, Chia-Che Tsai, Anurag Khandelwal, Qifan Pu, Vaishaal Shankar, Joao Carreira, Karl Krauth, Neeraja Yadwadkar, Joseph E. Gonzalez, Raluca Ada Popa, Ion Stoica, and David A. Patterson. Cloud programming simplified: A berkeley view on serverless computing. *CoRR*, abs/1902.03383, 2019.

[KBBK22] Gertrude Kappel, Christian Brecher, Matthias Brockmann, and István Koren. Internet of production: entering phase two of industry 4.0. *Commun. ACM*, 65(4):50–51, 2022.

[KS24] Anisha Kumari and Bibhudatta Sahoo. ACPM: adaptive container provisioning model to mitigate serverless cold-start. *Clust. Comput.*, 27(2), 2024.

[LGC+22] Zijun Li, Linsong Guo, Jiagan Cheng, Quan Chen, Bingsheng He, and Minyi Guo. The Serverless Computing Survey: A Technical Primer for Design Architecture. *ACM Comput. Surv.*, 54(10s), sep 2022.

[LKRL19] Junfeng Li, Sameer G. Kulkarni, K. K. Ramakrishnan, and Dan Li. Understanding open source serverless platforms: Design considerations and performance. In *Proc. the 5th International Workshop on Serverless Computing (WOSC@Middleware)*, pages 37–42. ACM, 2019.

[LLN+18] Eric Liang, Richard Liaw, Robert Nishihara, Philipp Moritz, Roy Fox, Ken Goldberg, Joseph Gonzalez, Michael Jordan, and Ion Stoica. RLlib: Abstractions for distributed reinforcement learning. In Jennifer Dy and Andreas Krause, editors, *Proc. 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 3053–3062. PMLR, 10–15 Jul 2018.

[LN24] Kexin Li and Stefan Nastic. Attentionfunc: Balancing faas compute across edge-cloud continuum with reinforcement learning. In *Proceedings of the*

13th International Conference on the Internet of Things, IoT '23, page 25–32, New York, NY, USA, 2024. Association for Computing Machinery.

[LSW+22] Jiewu Leng, Weinan Sha, Baicun Wang, Pai Zheng, Cunbo Zhuang, Qiang Liu, Thorsten Wuest, Dimitris Mourtzis, and Lihui Wang. Industry 5.0: Prospect and retrospect. *Journal of Manufacturing Systems*, 65:279–295, 2022.

[MKS+15] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A. Rusu, Joel Veness, Marc G. Bellemare, Alex Graves, Martin A. Riedmiller, Andreas Kirkeby Fidjeland, Georg Ostrovski, Stig Petersen, Charlie Beattie, Amir Sadik, Ioannis Antonoglou, Helen King, Dharshan Kumaran, Daan Wierstra, Shane Legg, and Demis Hassabis. Human-level control through deep reinforcement learning. *Nature*, 518:529–533, 2015.

[MNW+18] Philipp Moritz, Robert Nishihara, Stephanie Wang, Alexey Tumanov, Richard Liaw, Eric Liang, Melih Elibol, Zongheng Yang, William Paul, Michael I. Jordan, and Ion Stoica. Ray: A distributed framework for emerging AI applications. In Andrea C. Arpaci-Dusseau and Geoff Voelker, editors, *13th USENIX Symposium on Operating Systems Design and Implementation, OSDI 2018, Carlsbad, CA, USA, October 8-10, 2018*, pages 561–577, 2018.

[MPB24] Felix Moebius, Tobias Pfandzelter, and David Bermbach. Are unikernels ready for serverless on the edge? In *Proc. IEEE International Conference on Cloud Engineering (IC2E)*, pages 133–143, 2024.

[MSD+19] Anup Mohan, Harshad S. Sane, Kshitij Doshi, Saikrishna Edupuganti, Naren Nayak, and Vadim Sukhomlinov. Agile cold starts for scalable serverless. In Christina Delimitrou and Dan R. K. Ports, editors, *Proc. 11th USENIX Workshop on Hot Topics in Cloud Computing (HotCloud 2019)*, 2019.

[OPC24] OPC Foundation. OPC Unified Architecture: Interoperability for Industrie 4.0 and the Internet of Things. OPC official brochure, October 2024.

[Pla22] Aske Plaat. Deep reinforcement learning. *CoRR*, abs/2201.02135, 2022.

[Pri21] Gilles Privat. Guidelines for modelling with NGSI-LD. White Paper 42, ETSI, March 2021.

[RND23] Philipp Raith, Stefan Nastic, and Schahram Dustdar. Serverless edge computing - where we are and what lies ahead. *IEEE Internet Comput.*, 27(3):50–64, 2023.

[SB20] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning*. 2020.

[Sch22] David Scherer. A reinforcement learning approach for task execution in multi-provider faas, April 16th, 2022. Bachelor's Thesis.

[SQAS16] Tom Schaul, John Quan, Ioannis Antonoglou, and David Silver. Prioritized experience replay. In *Proc. 4th International Conference on Learning Representations (ICLR)*, 2016.

[SSH+18] Emiliano Sisinni, Abusayeed Saifullah, Song Han, Ulf Jennehag, and Mikael Gidlund. Industrial internet of things: Challenges, opportunities, and directions. *IEEE Trans. Ind. Informatics*, 14(11):4724–4734, 2018.

[SSK+21] Johann Schleier-Smith, Vikram Sreekanti, Anurag Khandelwal, João Carreira, Neeraja Jayant Yadwadkar, Raluca Ada Popa, Joseph E. Gonzalez, Ion Stoica, and David A. Patterson. What serverless computing is and should become: the next phase of cloud computing. *Commun. ACM*, 64(5):76–84, 2021.

[SWlT+23] Emilian Simion, Yuandou Wang, Hsiang ling Tai, Uraz Odyurt, and Zhiming Zhao. Towards seamless serverless computing across an edge-cloud continuum. In *Proc. IEEE/ACM 16th International Conference on Utility and Cloud Computing (UCC)*, 2023.

[WD92] Christopher J. C. H. Watkins and Peter Dayan. Q-learning. *Mach. Learn.*, 8:279–292, 1992.

[WLZ+18] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking behind the curtains of serverless platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 133–146, Boston, MA, July 2018. USENIX Association.

[WW24] Alexandre Gustavo Wermann and Juliano Araujo Wickboldt. KTWIN: A serverless kubernetes-based digital twin platform. *CoRR*, abs/2408.01635, 2024.

[XGT+23] Renchao Xie, Dier Gu, Qinqin Tang, Tao Huang, and Fei Richard Yu. Workflow scheduling in serverless edge computing for the industrial internet of things: A learning approach. *IEEE Trans. Ind. Informatics*, 19(7):8242–8252, 2023.

[YIWL21] Hanfei Yu, Athirai A. Irissappane, Hao Wang, and Wes J. Lloyd. Faasrank: Learning to schedule functions in serverless platforms. In *2021 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*, pages 31–40, 2021.

[ZBPG22] Haidong Zhao, Zakaria Benomar, Tobias Pfandzelter, and Nikolaos Georgantas. Supporting multi-cloud in serverless computing. In *Proc. 15th IEEE/ACM International Conference on Utility and Cloud Computing (UCC)*, pages 285–290. IEEE, 2022.

[ZWZ+25] Li Zhang, Cheng-Xiang Wang, Zihao Zhou, Yuxiao Li, Jie Huang, Lijian Xin, Chun Pan, Dabo Zheng, and Xiping Wu. Wireless channel measurements and characterization in industrial iot scenarios. *CoRR*, abs/2501.03577, 2025.