electronics

MDPI

Article

Cost and Latency Optimized Edge Computing Platform

István Pelle [1,2,*], Márk Szalay [1], János Czentye [1], Balázs Sonkoly [1] and László Toka [1,2]

[1] MTA-BME Network Softwarization Research Group, Faculty of Electrical Engineering and Informatics, Budapest University of Technology and Economics, 1117 Budapest, Hungary; szalay@tmit.bme.hu (M.S.); czentye@tmit.bme.hu (J.C.); sonkoly.balazs@vik.bme.hu (B.S.); toka.laszlo@vik.bme.hu (L.T.)

[2] HSN Labor Kft., 1036 Budapest, Hungary

* Correspondence: pelle.istvan@vik.bme.hu; Tel.: +36-1-463-3884

Abstract: Latency-critical applications, e.g., automated and assisted driving services, can now be deployed in fog or edge computing environments, offloading energy-consuming tasks from end devices. Besides the proximity, though, the edge computing platform must provide the necessary operation techniques in order to avoid added delays by all means. In this paper, we propose an integrated edge platform that comprises orchestration methods with such objectives, in terms of handling the deployment of both functions and data. We show how the integration of the function orchestration solution with the adaptive data placement of a distributed key–value store can lead to decreased end-to-end latency even when the mobility of end devices creates a dynamic set of requirements. Along with the necessary monitoring features, the proposed edge platform is capable of serving the nomad users of novel applications with low latency requirements. We showcase this capability in several scenarios, in which we articulate the end-to-end latency performance of our platform by comparing delay measurements with the benchmark of a Redis-based setup lacking the adaptive nature of data orchestration. Our results prove that the stringent delay requisites necessitate the close integration that we present in this paper: functions and data must be orchestrated in sync in order to fully exploit the potential that the proximity of edge resources enables.

Keywords: cloud native; edge computing; serverless; lambda; greengrass; FaaS; Function-as-a-Service; distributed data store; data locality; Redis

check for updates

Citation: Pelle, I.; Szalay, M.; Czentye, J.; Sonkoly, B.; Toka, L. Cost and Latency Optimized Edge Computing Platform. *Electronics* **2022**, *11*, 561. https://doi.org/10.3390/electronics11040561

Academic Editor: Vijayakumar Varadarajan

Received: 11 January 2022
Accepted: 9 February 2022
Published: 13 February 2022

Publisher's Note: MDPI stays neutral with regard to jurisdictional claims in published maps and institutional affiliations.

Copyright: © 2022 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (https://creativecommons.org/licenses/by/4.0/).

1. Introduction

Cloud computing is widely used in the digital industry as a technology that enables cheap and easy deployment not only for online web services but also for big data processing, Industry 4.0 and Internet of Things (IoT) applications. The public cloud is supported by physical data centers around the world that host virtual machines that are offered to customers and users. The cloud concept empowers the user to replace their own hardware with cloud server instances and creates a new economic model where the customer pays only for the usage and not for the entire hardware itself. As a reverse trend, the remote, central cloud is often extended, with private clouds and edge resources providing execution environments close to the users in terms of latency, e.g., in mobile base stations. Latency-critical functions can be offloaded from central clouds to the edge, enabling, e.g., critical machine-type communication or real-time applications with strict delay bounds. As a result, novel types of services and distributed applications can be realized on top of the edge or fog environment [1]. Such an application domain is transport, covering services such as automated driving, vehicle-to-vehicle communications, driver assistance, etc., which can reshape our digital society and, indeed, necessitate an edge platform that provides apt solutions for the latency-critical and data-intensive application requirements amid the challenges of user mobility and the spatial and temporal dynamics of the application load. Besides enabling infrastructural spread, i.e., edge nodes are potentially closer to the end users, we argue that an edge computing platform must be locality-aware and

proactive in terms of resource provisioning in order to further decrease end-to-end latency for application users. We envision a dynamic system that allows for the migration of service components and complete re-optimization of their placements periodically.

The virtualization techniques and the value-added services in the cloud demonstrate a wide variety today to meet the heterogeneous customer demand. One can list pros and cons for all types of cloud services: there is a diminishing burden on cloud tenants in terms of application management from Infrastructure-as-a-Service (IaaS) through Container-as-a-Service (CaaS) to Function-as-a-Service (FaaS), although this, of course, comes at an increased price of resource units; built-in features and their resource usage are incorporated into these prices. Adopting the stateless design in its core concept [2], FaaS, often referred to as serverless computing, has recently become one of the most popular paradigms in cloud computing. The paradigm emerged not only as a pricing technique but also as a programming model promising to simplify development for the cloud. Using FaaS, developers do not need to take into consideration resource allocation, scaling or scheduling, since the platform handles these. Numerous projects managed by companies and academic institutions have built FaaS platforms, but the most widely used ones are underneath the FaaS services offered by IT giants, i.e., Amazon's AWS Lambda [3], Google Cloud Functions [4] and Microsoft Azure Functions [5]. Most of these platforms operate with container technologies; the user's executable code is packed into a container that is instantiated when the appropriate function call request first arrives. With this relatively lightweight technology, it is easy to achieve process isolation and resource provisioning.

FaaS systems usually accommodate ephemeral functions and their load is hectic; hence, their function placement logic to be applied must be fast and efficient. Function requests arrive frequently and demand various resource amounts and a great level of elasticity. Therefore, by its nature, the FaaS service platform must apply an online placement method, probably with affinity constraints to consider in order to collocate functions that may invoke each other [6]. Inherent to the FaaS concept, input data or internal function states are often externalized—hence, the stateless operation. As network delay might cause serious QoS degradation when remote data must be accessed by the functions invoked in the FaaS platform, the placement of these is of paramount importance too [2,7].

In this paper, we advocate for the joint placement of functions and their respective states in edge systems. Our contribution is three-fold. (i) We showcase an integrated platform that simultaneously provides cost-optimal, latency-aware placement of FaaS functions and access pattern-aware data relocation. Our platform builds on our previous prototypes, implementing an application layout optimization and deployment framework [8] and a distributed key–value store [7], respectively. (ii) We evaluate this integrated edge platform through a comparison focusing on data access in our adaptive data store [9] and in Redis [10] for use-cases where heavy user mobility is assumed across edge sites. (iii) We show that the access patterns generated by our autonomous transport-inspired use-cases call for the need of synchronization between function and data placement and that our integrated solution achieves better performance, leveraging this functionality. Our goal with this work is to demonstrate that a network delay-aware orchestration policy achieves lower end-to-end delays for cloud applications that serve users moving frequently from one edge site to another. Our presented platform not only adapts the location of computational entities across the cloud continuum to the actual geographical demand, but also that of the related data. Opposed to the composition and placement of computational entities that is driven by a cost optimization target while considering application latency requirements as constraints, the orchestration of the application-related data is driven by delay minimization as its ultimate objective.

Our work is motivated by the new era of the IoT that is driving the evolution of conventional vehicle ad hoc networks into the Internet of Vehicles (IoV). IoV promises huge commercial interest, thereby attracting a large number of companies and researchers [11]. As the number of connected vehicles keeps increasing, new requirements (such as seamless, secure, robust, scalable information exchange among vehicles, humans and roadside

infrastructures) of vehicular networks are emerging [12]. We argue that, first and foremost, the stringent delay requirements must be met in IoV—hence, the motivation for our edge platform.

This paper is organized as follows. In Section 2, we introduce the transport application scenarios for which our proposed edge computing platform provides the necessary features, i.e., compute resource and data orchestration. We give an overview of the research work related to such features in Section 3. Building on our prior work, we provide a description of our integrated edge platform in Section 4. Then, Section 5 delves into the quantitative performance evaluation of our proposed platform in terms of data access delay of FaaS instances associated with users that are moving from the vicinity of one edge site to another. Finally, in Section 6, we summarize our findings and conclude the paper.

2. Use-Cases Involving User Mobility

In classical IoT use-cases, sensors infrequently wake up and send usually small quantities of data. The challenge of processing these data arises not from the latency-sensitiveness of the applications but rather the myriad IoT devices and consequently the amount of data to be processed. In Industry 4.0 use-cases, data processing and altogether application delay becomes a more pressing issue as control processes might need to take action with stringent latency requirements. Usually, these use-cases do not require the frequent relocation of data processing tasks because the data sources are fixed or their mobility is limited.

With autonomous driving and automated transport ever approaching, the automotive industry is shifting towards being software-driven as well. Continental, one of the most prominent suppliers of automotive parts, envisions an increase in on-board computing power to enable cross-domain high-performance computing [13] to handle highly latency-sensitive tasks over a unified computing platform. With the help of Amazon Web Services (AWS) [3], the company has started to lean on cloud processing as well, albeit currently only in aiding software design, testing and validation [14]. As this example shows, processing might be done more cost-efficiently in data centers; thus, cloud technologies have already started to find their way into automotive scenarios to enable faster development and better maintenance.

In certain cases, however, such low latency is required that processing cannot be offloaded to cloud data centers but needs to be located close to moving data sources. As vehicular movements show strong geographical dynamics and can span countries or even continents, edge processing proves to be adequate in such situations. Since vehicle traffic is dynamic in nature, edge platforms need to meet requirements of high and fast scalability. In recent years, FaaS solutions emerged as options for handling small processing tasks with high dynamicity by providing fast request-based scaling features. This concept builds on small stateless functions that might hinder transport use-cases that leverage historical data, e.g., previous positions of nearby vehicles in order to estimate their trajectory or speed in an automated overtake control scenario. While externalizing states solves the issue of stateless functions in scenarios where computation tasks are always bound to fixed processing nodes, in automated driving scenarios, this is not the case. Here, processing tasks need to follow data sources over large geographical areas in order to always provide low latency. This implies that data need to be synchronized or relocated among edge nodes to provide the seamless transition of tasks between nodes.

We argue that in order to serve such use-cases, a platform needs to offer highly scalable computing resource allocation and adequate data synchronization or relocation features. We present our proposal for such an integrated platform in Section 4.

3. Related Work

In this section, we give a literature survey about current computing resource optimization and placement techniques and tools, as well as summarizing cloud and edge data storage options.

3.1. Cost and Latency Optimization in Edge/Cloud Scenarios

In contrast to our platform proposed in this paper, to the best of our knowledge, currently, there is no edge computing platform that altogether deals with (i) the latency of the deployed applications, (ii) the cost of running the application in the edge/cloud, (iii) the exact location of application data within the platform and (iv) the mobility of the users. However, there exist multiple solutions [15–22] that consider and examine three out of these four features.

The authors of [15] assume a single-operator scenario in a multi-cloud system, in which they provide an offline mechanism to place the application elements among the cloud data centers. Besides the delay and revenue, the device power consumption is also targeted in terms of optimization objectives. Researchers in [16] suggest game-theoretic techniques for virtual machine (VM) placement in edge and fog computing systems to ensure the application's performance by applying mobility patterns, while they aim to jointly minimize infrastructure energy consumption and cost. Ref. [17] studies a multi-edge infrastructure for which the authors formulate the edge device placement problem as a virtual network function (VNF) placement task for reliable broadcasting in 5G radio access networks (RAN). The problem is formulated as a multi-objective optimization problem constraining bandwidth, service latency and processing capacity and minimizing the composite objective function for reliability, deployment cost and service response time. The particle swarm optimization and genetic algorithmic meta-heuristic approaches are used to solve the optimization problem.

The research work [18] focuses on the customers' aspects and optimizes the user experience. It addresses the offloading of distributed applications (performance-sensitive IoT applications) constructed by multiple connected (or, more generally, related) components in a cloud-edge-terminal scenario. Besides the delay, the revenue is also part of the optimization objective, and they consider bandwidth, computation and mobility in the optimization problem as constraints. Ref. [19] considers the underlying infrastructure, where the service components are mapped to, as a multi-edge computation system. The novelty of this work lies within the aspect of application migration to optimize the system's delay and utilization properties.

The study [20] focuses on the medium-term planning of a network in an MEC environment. The authors define a link-path formalization along with a heuristic approach for the placement of virtualization infrastructure resources and user assignments, i.e., determining where to install cloudlet facilities among sites, and assigning access points, such as base stations, to them. They calculate the mobility patterns and migrations and optimize them for the revenue. The authors of [21] propose a system that supports mobile multimedia applications with low latency requirements. Their study targets the problem of the dynamic placement of service-hosting nodes over a software-defined network (SDN)-based, NFV-enabled MEC architecture to minimize operational costs. They focus on satisfying the service level response time requirements. To this end, they present an online adaptive greedy heuristic algorithm, which is also capable of managing the service elasticity overhead that comes from auto-scaling and load balancing with a proposed capacity violation detection mechanism. Ref. [22] recognizes the challenge caused by migrations in a multi-edge infrastructure. It proposes an energy-aware optimization scheme that minimizes the latency and the involved reallocation costs due to the limited edge server budget and user mobility.

In recent years, further articles have been published on the topics of network function virtualization (NFV) [23,24] and service function chaining (SFC) [25–28], which typically provide applicable heuristics and methods for utilizing the virtualized resources of the underlying cloud and edge infrastructures in an efficient way. These specific algorithms utilize a wide variety of mathematical apparatus to determine the optimal placement of the application components, considering different constraining factors and optimization objectives, such as utilization, power consumption, revenue or application latency [29]. One of the crucial features of these placement methods is that they require topological

information about the provider's infrastructure, consisting of the available computational, storage and network resources and the defined interconnections between them. Moreover, these topology models usually include accurate and up-to-date information about the low-level characteristics of these resources, such as the speed of the CPU, number of cores, networking delay, etc., and their utilization, as well.

However, with the novel concepts of cloud-native and serverless computing [30,31], the operational burdens of the application components are mostly taken off the shoulders of the service maintainers. These paradigms also imply that the exact location of these components (in the provider's cloud), the allocated CPU types and the runtime utilization of the underlying shared hardware are all unknown before the initiation and scheduling of the related artifact units. The lack of relevant topological information prevents the leveraging of any placement algorithm for cloud-native service optimization and hinders the efficient operation of latency-sensitive applications over today's cloud and edge platforms. Therefore, novel methods are required along with the specific service descriptors and cloud platform models to tackle the emerging optimization challenges in serverless computing.

3.2. Cloud/Edge Data Storage Solutions

In the case of geographically distributed computing systems, there exist multiple choices for a highly configurable and dynamic key–value store that aim to provide a data layer where applications can externalize their internal data. We find that, in terms of the data access time minimization, we can distinguish them according to the following aspects:

- Is it possible to make copies of the stored data?
- Does the storage optimize the locations of the copies or even the original data within the storage cluster?
- Is making replications supported at node or data level?
- Does the storage support some kind of data access acceleration technique?

Popular key–value stores such as Redis [10], Memcached [32], Cassandra [33] and DynamoDB [34] use consistent hashing or a hash-slot/node association for data placement, which, in terms of the data access time, is a random placement approach. If the data are co-located with their reader and writer applications, the access time will be quite low due to the lack of inter-node access. However, if the data are located on a different node, we have to calculate with the additional network delay as well. The known solutions above—by the random data placement—allow fast data lookups; however, they do not tackle the minimization of remote state access. Moreover, data can be shared among multiple reader or writer applications. In this case, the data can be replicated on multiple nodes to avoid inter-node readings (or writings). The above-mentioned key–value stores support a fixed data replication factor strategy, which means that all data will be replicated in the same number; there is no difference between "hot" and "cold" data. Finally, in order to accelerate the data access, they use cache solutions in the memory.

There are some approaches from academia as well. Anna [35] is a distributed key–value store that uses a hash function to calculate the locations of the stored data within the cluster, but it enables selective replication of hot keys to improve the access latency. DAL [2] allows data access pattern-aware data placement, i.e., it handles individual data items independently and attempts to find them the optimal target server to minimize the access latency. Furthermore, it uses replication factors for each data entry individually. As a data access acceleration technique, for each data entry, it maintains a list of servers where access requests arrive and moves the data to the one generating the highest amount of queries.

3.3. Automated Deployment to Cloud/Edge Infrastructure

Different tools exist that can determine and automatically configure the required cloud resources in order to ease the complex task of cloud application deployment across different platform service providers. For instance, the Serverless Framework [36] offers a provider-agnostic approach to declare and manage resources over public clouds, using its own CLI interface and a YAML-based descriptor file. Similarly, Terraform [37] provides a

higher-level interface and a toolset to set up and manage cloud infrastructures spanning over multiple public cloud domains. It allows customers to seamlessly migrate their whole virtualized infrastructure from one provider to another by hiding most of the provider-specific configuration parameters. There are other tools specifically designed to apply external parameters from other services at deployment time, such as specifying resource type or memory size. One such external service is Densify [38], which can make cloud applications self-aware by leveraging its separate optimization and monitoring components. It can collect CPU, memory and network utilization data about virtual machines in Amazon with proprietary monitoring services, which then can be leveraged by its machine-learning-empowered optimization component to model the application's utilization patterns, estimate the optimal compute resources and give recommendations on the type and number of instance flavors for the application maintainer. In addition, it provides automatic redeployment features relying on templating tools that support dynamic parameter assignment or parameter stores.

Amazon also offers different services for managing cloud resources, which rely on the common AWS API but realize functions over diverse complexity levels. In contrast to low-level options, i.e., the web console, SDKs and the CLI, which are only suitable for simpler resource management, the widely used AWS CloudFormation [39] can treat a whole deployment as a unit of workload in the form of stacks or stack sets. It can handle the setup, modification and deletion tasks of complex applications by way of its proprietary templating language. Stackery [40] is another tool designed to accelerate the deployment and operation of complex serverless applications through their life-cycle, on top of AWS's cloud.

Although the aforementioned tools provide a wide range of assets to manage complex serverless applications over public clouds, they lack the support of hybrid edge-cloud deployment scenarios where application latency is of concern. While AWS offers tools for managing edge nodes, and its Compute Optimizer service [41] works as a recommendation engine that helps right-sizing EC2 instances, none of the services offered by AWS can handle optimization tasks for serverless applications. Other similar tools that consider utilization for serverless resources also neglect to take application performance into account, while they either completely lack edge infrastructure handling capabilities or introduced similar features only recently. Besides these tools, only a few academic research papers consider the cost-aware modeling and composition of serverless applications. The authors of [42–44] propose pricing models and cost analysis for serverless deployment scenarios but in an offline manner. Costradamus [45] offers online, per-request cost tracing features to overcome the limitation of high-level billing schemes by leveraging a fine-grained cost model for the deployed cloud application; however, there is no optimization support. Other recent works [46–50] study cloud-native performance optimization over public clouds, but they are missing any adaptation feature or automated application re-optimization.

4. The Proposed Edge Platform

Our system offers an integrated platform that enables four main features. First, it is able to calculate the cost-optimal grouping and compute resource size assignment and the placement of application components while still adhering to application end-to-end latency constraints. Second, it is capable of deploying the tasks according to these considerations onto edge and cloud resources. Third, it optimizes the placement of externalized data accessed by the application components to provide the lowest achievable data access latency. Fourth, it continuously monitors application performance and considers recent measurements in the component optimization algorithm.

The platform takes advantage of our previous works exploring application deployment to cloud and mainly edge resources [8,51] and access pattern-aware data storage [9], achieving synergies for providing better application performance. Figure 1 shows a high-level view of our system. At the top level, the application developer or maintainer supplies two sets of data regarding the application itself: the source code and the description of

application invocation properties. In the second case, the developer defines the ingress point of the application, as well as invocations and their rates among the components. The exact method of discovering these aspects is outside of the scope of our current work, but data can come from baseline measurements or external code analyzer tools. At this point, the developer can also set an upper limit for the application's end-to-end latency. Automatic application deployment is performed in three stages. First, the Layout and Placement Optimizer (LPO, discussed further in Section 4.1) determines application function grouping, resource location and sizing utilizing the invocation properties and respecting the given end-to-end latency constraint. Second, the Serverless Deployment Engine (SDE) prepares deployable artifacts from the application code using the software layout given by the LPO as a blueprint. These artifacts are then used for setting up the application itself and the performance evaluation service. A unified edge-cloud management API is leveraged to perform the actual deployment tasks to cloud and edge resources. Once deployed, the application's performance is monitored by a dedicated component that can alert the LPO or the Offload Controller (OC) when performance criteria are not met. Based on performance metrics that are not met, the LPO can recalculate the optimal resource assignment and initiate the redeployment of the application. The OC, on the other hand, can perform limited application reconfiguration and move application components between predetermined placement locations.

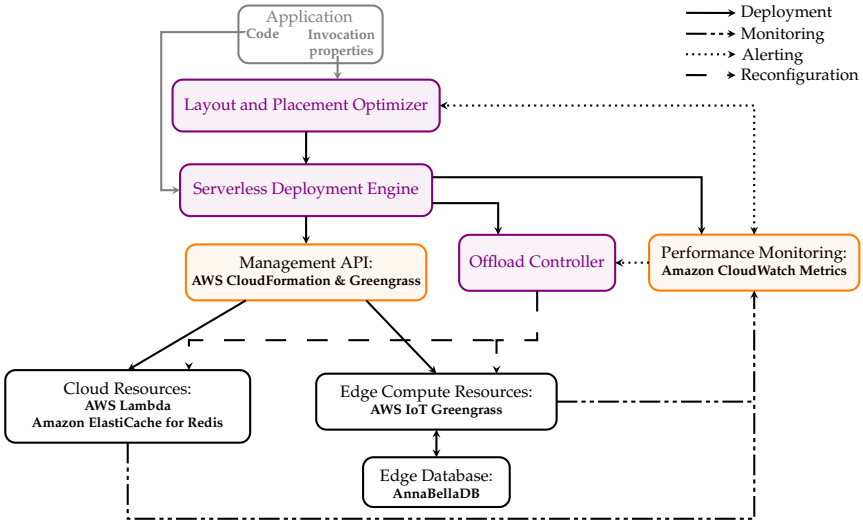

Figure 1. High-level overview of the application layout optimization, deployment and monitoring system with database options.

To realize our general concept discussed above, we rely on AWS and its multitude of services in cloud and edge computing and monitoring. We leverage a Function-as-a-Service offering, AWS Lambda [52], in the cloud, and Greengrass [53], its counterpart at the edge. Utilizing these provides us with a unified execution runtime for application components. The serverless nature of these services offers low management overhead and highly scalable request-based on-demand resource allocation that carries benefits even for edge deployments. AWS Lambda grants predefined runtime environments for different languages and fine-grained memory and CPU allocation schemes. A function instance is started when there is no other instantly available instance to serve an incoming request. This first start-up time is usually slower than all subsequent calls to the instance as, during warm-up, certain parts of the user code are initialized and cached for later invocations. In the cloud, function instances are terminated after a provider-calculated period of time being idle. At the edge, they are kept warm until resource utilization at the edge node allows it. Amazon CloudWatch Metrics is selected to monitor and store performance data coming in from the application and edge infrastructure and provide alerting capabilities.

As our chosen application execution platform offers runtime for stateless components, state externalization is a fundamental requirement. In order to support frequent state access at scale, a highly efficient storage option is needed. In cloud-only deployment scenarios, ElastiCache for Redis is chosen as our previous performance evaluation [54] shows that it offers low-latency data access even at high access rates and data sizes. In edge scenarios, however, using a data store located in the cloud is not an option anymore because of the possibly long delay between edge nodes and cloud data centers. In Section 5, we showcase that selecting Redis in an edge-only deployment scenario where multiple nodes are used can be disadvantageous, and using an option that can adaptively relocate data based on the access location and frequency is a more viable alternative. We give a short introduction to AnnaBellaDB [9], our choice of data store in multi-edge node setups, in Section 4.2.

4.1. Computing Optimization

Our platform's Layout and Placement Optimizer (LPO) component builds on our prior works [8,55]. In these, we discuss the optimization problem of cloud-native service composition over hybrid edge and cloud infrastructures, focusing on Amazon's serverless offerings, i.e., AWS Lambda [52] and AWS IoT Greengrass [53]. Stated as the opposite problem of decomposing a monolithic application into a microservice architecture, service composition is the task of designating the main application components as standalone deployable artifacts by selecting and grouping the basic building blocks of the application, i.e., the cloud-native functions, in a bottom-up manner [56]. In addition to the function grouping, it is also required to assign the appropriate runtime flavors, i.e., defining the amount of resources for the application components. Therefore, one possible realization of a cloud-native application can be described with the groups of compiled and assembled functions along with the related flavor assignments, datastore selection and related deployment configurations, which are collectively referred to as the application layout. The main observation regarding the feasible layouts of an application is that different partitioning (also called clustering, equivalently) of the functions' call graph results in different overall execution times based on the invocation patterns and involved invocation and data access delays, but with diverse operational cost implications due to the runtime-based pay-as-you-go billing schemes. In the case of combined edge/cloud infrastructures, the varying capabilities, platform characteristics and cost implications of the central cloud and dispersed edge nodes also need to be taken into account during the function partitioning process. Moreover, latency-sensitive applications usually impose stringent latency requirements, either in an end-to-end manner or on subchains of the given application [57], which complicates further the selection of the cost-optimal layout. According to these aspects, the main goal and the outcome of the optimization task, which is performed by our LPO module shown in Figure 1, is to find the cost-optimal service layout over the given capabilities of the provider's cloud/edge infrastructure, while the user-specified latency requirements are met.

For describing such cloud-native applications, the main call structure of the building blocks is required to be formed as a directed acyclic graph (DAG). The nodes in the service graph denote the basic functions, which can be invoked by only one other function, and also a single datastore entity, whereas the arcs mark the read/write data accesses and function invocations. Functions as stateless, single-threaded building blocks are characterized with their reference execution time measured on 1 CPU core, while the arcs are described with the average invocation rate and the blocking delay introduced in the invoker function. The capabilities of execution environments are described by the general notion of flavors, where edge nodes have distinct flavors. According to our previous performance studies in [54,55], cloud resource flavors are specified by their offered vCPU fractions, as, for single-threaded functions, the assigned flavor size is directly proportional to the computational performance. This trend holds until reaching the peak value at 1769 MB, when one CPU core is granted. The operational cost characteristic of each type of platforms, i.e., the central cloud and the distinct edge nodes, is realized via a single dedicated cost factor. This factor,

by definition, describes the unified cost of code execution given for a predefined unit of time, which is typically 100 ms in Amazon's cloud [58]. These factors are used for the cost calculation of application components in feasible layouts, relying on the platform-specific execution time of the components.

Our algorithms proposed in [8,55] use a recursive, dynamic programming-based approach to define the minimal cost application partitioning by dividing the function invocation DAG into groups of constituent functions. The bottom-up methods iteratively derive the actual execution time of feasible partition blocks based on the serialized execution of the encompassed functions, the invocation ratios, blocking delays and the assigned flavor's vCPU fraction, while invocation delays within a group are considered negligible. The group costs are calculated using cost factors corresponding to the cloud provider's pay-as-you-go billing model. For the latency calculations and requirement validations, an adjusted execution time formula is applied to correctly match the measured latency metrics in the performance monitoring system. In detail, our proposed algorithm is divided into two main procedures. The chain partitioning task can calculate the cost-optimal grouping of a chain of functions, which can have latency constraints defined on disjoint parts of the chain. This algorithm recursively decomposes the chain partition problem into subproblems or subchains regarding the different latency limits, while the optimal grouping and cost/latency values of the subproblems are calculated by relying on previously calculated sub-solutions that are stored in dedicated dynamic programming matrices. The algorithm calculates the optimal solution of the subchains starting from the beginning of the chain and considers the following component in each iteration, until the cost-optimal solution of the original application chain is found in the last step. The tree partitioning is a generalized algorithm of the serverless application partitioning problem, where the main application as a DAG is also decomposed into subproblems. Our algorithm, following a similar recursive concept, can find the optimal solution by decomposing the DAG into chains of the constituent functions while considering the latency constraints, and it utilizes our chain partition algorithm to acquire the cost-optimal solutions for its subproblems in a bottom-up manner. It also applies several subtasks, such as pruning a distinct chain from a subtree or ensuring the latency constraints on the critical paths of the DAG so as to comply with the preconditions of dynamic programming. Pseudo-codes and exact formulas explaining the algorithm are detailed in our previous work [8], along with the related time complexity evaluations.

4.2. Data Optimization

In geographically distributed systems, such as the edge computing platforms, the latency of user applications depends not only on the computation platform and the network but also on the placement of the application data within the cluster. Unless the application data are not close to the end user, the smart location of the function is in vain, e.g., let us assume that a smart car is connected to the nearest mobile base station, where—on an edge computation node—a 'road condition reporter' application is located. If the road condition data are stored in the edge node, during the road status query, the car experiences only the vehicle-to-edge communication latency and vice versa. On the other hand, if the status data are located in a central cloud, the query latency also includes the additional delay between the edge node (where the function is running) and the cloud database (where the data are located). One might imagine, e.g., in the case of slippery roads, that this road information's delay may significantly influence the chances of a safe passage.

There are multiple approaches for organizing application data within a distributed system, both from industry [10,34] and academia [2,9,35]. The solutions differ in countless dimensions, e.g., data placement methods, data replication management, used data models or data access acceleration techniques. One of the major research questions—in terms of the data access latency—is whether the data store determines the data locations based on some form of hash function or whether it is so-called *access pattern-aware*, i.e., the data placement is based on priority or on the monitored access properties of the stored data, such as the

number of reads and writes from different nodes. Another challenging task is to determine the optimal number of data copies with the goal of minimizing the access time, e.g., in the case of frequently read and rarely written data, each application should maintain a local copy of the original data, thus minimizing the read times. Furthermore, it is essential to calculate with the used data model in the edge cloud storage. Several solutions assume multi-parent (the original and the copies of each data instance are readable and writable), while others prefer parent–child types (the data copies are only readable) of data. The former results in increased data synchronization time in the case of parallel data writings, while unavoidable children update time comes with the latter data model.

For the platform presented in this paper, we propose to use our formerly published AnnaBellaDB [9] (ABDB) solution, which realizes the state layer [7] of the edge platform, where the user applications can externalize their operational data to. ABDB is an access pattern-aware key–value store that uses a parent–child data model and data-level replication handling. This approach is the opposite of Redis [10], one of the most popular key–value stores, since, instead of using range or hash partitioning for data placement, it continuously re-optimizes the data locality in the cluster if the change in data access has exceeded a predefined threshold. The architecture of ABDB is provided in Figure 2.

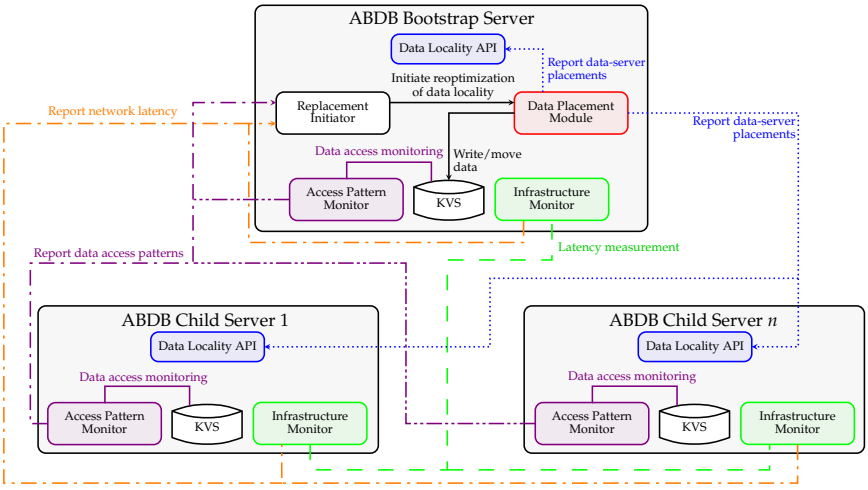

Figure 2. Architecture of AnnaBellaDB.

AnnaBellaDB is a distributed key–value store, i.e., it forms a database cluster containing one bootstrap server and multiple children nodes. To minimize data access, each edge server should include an ABDB server, either bootstrap or a child in an edge computing infrastructure. Each ABDB server contains an Infrastructure Monitor, which sends pings to other servers and measures the network latency between the edge nodes. As a result, they periodically report the measured latencies to the Replacement Initiator module that is located on the bootstrap server. Furthermore, all servers contain the actual key–value store (KVS), where the users' key–value pairs are stored. The user applications read from or write into these KVSs. Meanwhile, the Access Pattern Monitor modules continuously monitor the access patterns of the locally stored data, i.e., the number of readings and writings per data entry. Moreover, they report these access patterns to the Replacement Initiator. Should the access pattern of a datum change more than the predefined threshold, the Replacement Initiator—as its name suggests—initiates the movement of the monitored data to the new target server through the Data Placement Module. If an application wants to access a piece of data, first, it asks the co-located Data Locality API about which server stores it, and it then targets the KVS on the proper server. In the case of a first write, the Data Locality API always returns the address of the Data Placement Module, which determines which server will store the data. If a new writer or data movement happens, the Data Placement Module reports to the other servers in the cluster about the new data server assignment.

The Data Locality API components store and maintain the data distribution across the cluster, i.e., a list of data and its corresponding storage server (data, server) pairs. When the cluster is in a steady state, i.e., each Data Locality component has the same list about the cluster's data, all the functions obtain the same result about where to find the requested data, independently of where they are located. Thus, they will know which server should query for the given data. All Data Locality components are updated periodically by the bootstrap server. However, if a co-located Data Locality component does not know or knows poorly where the requested data are stored, it must retrieve the target server from the bootstrap server. Consequently, the child servers will ask the bootstrap server only at the first data request or when the queried data have been migrated and its local related query has not been updated yet.

Determining the optimal locations of a set of data to minimize their cumulative access latency is an $\mathcal{NP}$-hard problem [7]. However, as [59] argues, if the ABDB servers would have infinitely large capacities to store data, then calculating the optimal destinations within the server cluster could be solvable in polynomial time. Based on this theorem, we have created our heuristic placement solution in the Data Placement Module. First, we assume that each server has infinite capacities and calculate the optimal node for each data using their access patterns as inputs, i.e., the reader and writer functions, their locations and their access frequency. As a result of this first step, it might occur that the placement overloads a node with too much data to store. If this is the case, in the second step, the heuristic algorithm picks the data from the most loaded server, which is accessed the least from other nodes, and migrates them to the closest server, in which the free capacity is larger than that from where the data are moved. The algorithm repeats the second step as long as an overloaded server exists. The heuristic algorithm mentioned here is elaborated in [7]. AnnaBellaDB is open-source and available at https://github.com/hsnlab/annabellaDB (accessed on 28 November 2021).

One might note that there exist other data access acceleration techniques, e.g., *client-level caching*, which utilizes data caches in the applications besides a traditional key–value store such as Redis. However, it is essential to see that this approach is not applicable in the case of FaaS platforms since there is no one-to-one mapping of the clients and their functions within FaaS. For example, suppose that multiple workers of the deployed function live in the cluster waiting for the client's invocations. In this case, the actual worker that will execute the function is scheduled by the FaaS scheduler. Consequently, the data are unnecessarily cached in one function if, in the next round, another function will be activated for the same client's invocation. Instead, the node-level caching could work, in which the co-located functions have access to the node's storage. Our presented ABDB works similarly: each server has its own KVS, and the Replacement Initiator takes care of the data migrations. This results in specific data being stored on a server from which the majority of the access originates.

4.3. Application Deployment, Operation and Monitoring

Components responsible for application deployment and operation extend the capabilities of the tools shown in our previous works [8,51]. After the Layout and Placement Optimizer component calculates the optimal application layout, the Serverless Deployment Engine (SDE) together with the AWS infrastructure management tools deploys the application components to edge and cloud resources. Based on upper-level instructions, the SDE is able to combine multiple application components, functions, into a single AWS Lambda function. A crucial component that enables the creation of AWS Lambda functions in this way is the purpose-built Wrapper that is injected into the application code by the SDE. It encapsulates all application components and acts as an intermediary layer between the AWS Lambda/Greengrass runtime and the application component. As shown in Figure 3, every interaction between f_I application components, deployed as F_K AWS Lambda serverless functions, passes through the Wrapper, be this data store access or the invocation of another application component. The f_I application components have to interact only

with their Wrappers' unified interface to get in contact with the outside world. Two of the value-added features of the Wrapper exploit this encapsulation. One supports monitoring and sends actual invocation and data access rates, sizes and delays to Amazon CloudWatch in order to extend application performance monitoring and help troubleshooting. The other feature provides fast reconfiguration options. Here, an external Offload Controller (OC), which can be located either in the cloud [8] or at the edge [51], notifies the Wrapper to change the application layout on the fly without redeployment. In a scenario where application function f_i invokes f_j and they are assigned to different serverless functions, F_n and F_o, respectively, deployed to edge nodes A and B (as in Figure 3), the Wrapper can be instructed by the OC to divert all further $f_i \rightarrow f_j$ calls from edge node A to the other, effectively offloading f_j to node B. Leveraging Wrapper capabilities, we also circumvent the default AWS IoT communication methods when sending invocations from the edge to the cloud [8] and between edge nodes [51] to significantly reduce the communication latency.

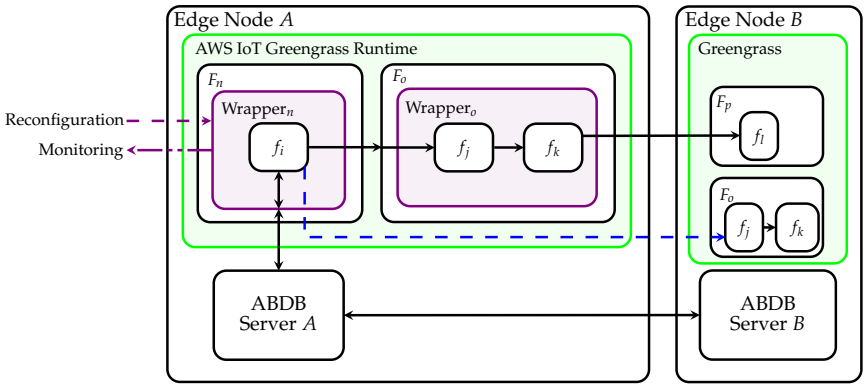

Figure 3. Simplified overview of application operation.

In the edge platform proposed in this paper, we extend the Wrapper's list of supported data stores (local and Redis) with AnnaBellaDB. The Wrapper connects to ABDB via its Python client library. When software layout blueprints arriving from the LPO indicate the use of ABDB, the f_I functions will use this option without any modification needed in their code. As the Wrapper is context-aware, it can configure access to ABDB child servers so that functions running on a certain edge node would have access to the ABDB child running on the same node, enabling low-latency access. Wrapper functionality also enables the logging of data store access performance to Amazon CloudWatch to provide monitoring and alerting capabilities.

In the next section, we exploit the above-mentioned monitoring and function offloading features to provide a comparison between the performance of AnnaBellaDB and Redis in scenarios where all application functions are deployed to edge nodes using our deployment framework.

5. Results

In order to evaluate the performance of our platform, we show two sets of comparison measurements in this section. In both cases, we deploy generalized benchmark applications to an edge environment using our deployment solution. The sample application is then operated by the AWS Greengrass runtime. Data store access to ABDB and Redis is realized by our Wrapper component. Using the monitoring capabilities of our platform, we record data read and write delays for both data stores and show that, in cases with large data sizes and multi-node, multi-function environments, data locality becomes a key factor in the application performance.

5.1. Size Dependency of Data Access Delay

In order to determine data access performance under varying data sizes, first, we set up a sample edge environment consisting of two edge nodes, as shown in Figure 4. A single-node setup of Redis is deployed on *Edge node 1* with the default configuration and password authentication enabled. For ABDB, a child server is set up on the same node and a bootstrap server on the other. Lenovo System x3550 M5 servers equipped with two physical Intel Xeon E5-2620 v3 CPUs, each having 12 cores running at 2.40 GHz, 64 GB memory and Broadcom BCM5719 quad-port 1GBASE-T Ethernet controllers are used as edge nodes. The nodes are connected through an HP 3500yl-24G PoE+ switch. Our sample application consists of a single function that is also deployed to *Edge node 1*. The single benchmark function performs one write operation of a random string value, followed by a read operation for both data stores. Evaluation is done for data sizes between 10 bytes and 1 MB, and operations are repeated 100 times with a warmed-up function instance. The data stores are reset between subsequent runs (used keys are deleted).

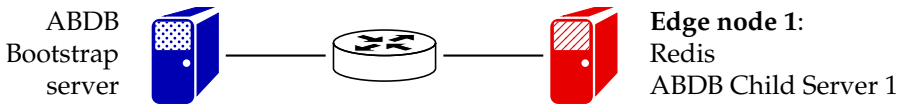

Figure 4. Experimental edge and network setup for evaluating single (child) node performance.

Figure 5 shows average, minimum and maximum read and write delays for every case. At low data sizes, the delay of accessing Redis is half of ABDB's because of the additional processing overhead of the latter, while, for larger chunks of data, the gap is smaller between the two options but Redis always remains the faster option. We note that, comparing Redis's performance to that of its cloud counterpart [54], we can observe that, at small data sizes, our edge deployment's performance falls between that of a *cache.t2.micro* (standard cache node with 1 vCPU, 0.555 GiB memory and low to moderate network performance) and a *cache.r5.large* (memory optimized node with 2 vCPUs, 13.07 GiB memory and up to 10 Gigabit network throughput) ElastiCache for the Redis instance. Writing 1 MB-sized data is slightly faster than with a *cache.r5.large* instance, but reading is still faster with *cache.r5.large* when a Lambda function is deployed in the cloud.

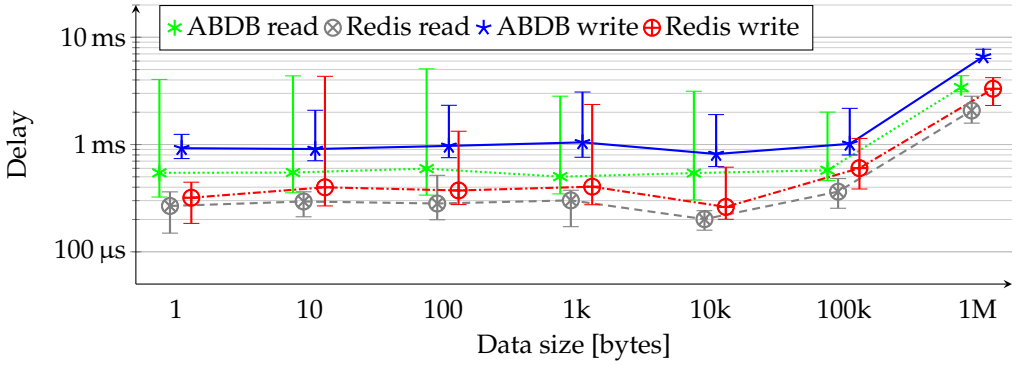

Figure 5. Data access latency of functions deployed to an edge node where an ABDB child and a Redis server are both available. Maximum, average and minimum values are shown.

We evaluate read and write performance in a simple remote access case on an extended edge setup, depicted in Figure 6, where we add a second ADBD child server. In this case, we deploy two benchmark functions: the first one is deployed on *Edge node 1* and performs an initial read and write operation and then calls the second function located on *Edge node 2* that accesses the specified key once per second, obtaining 100 samples, similarly as before. We note that, in Redis's case, this results in actual remote access, while ABDB relocates the key from *Edge node 1* and then accesses it locally on *Edge node 2*, giving the same average performance as seen in the previous case. Our measurements show that the round-trip time

between each node pair is around 0.155 ms on average with small data sizes. Round-trip time gradually increases due to network delay to around 0.4 ms at 10 kB and 1.6 ms at 65 kB as reported by ping.

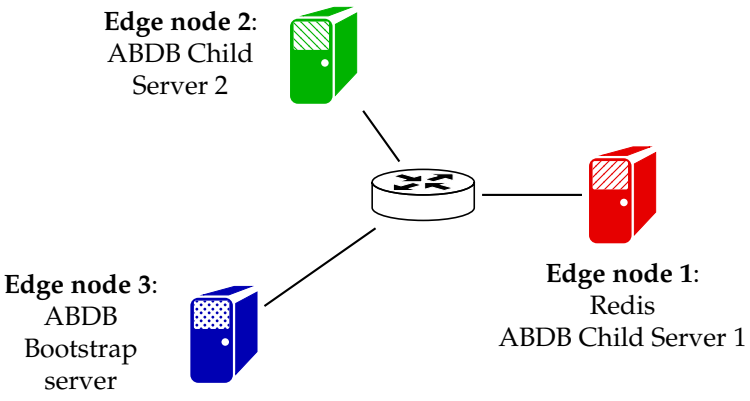

Figure 6. Experimental edge and network setup with three edge nodes.

Measurement results obtained during the tests, displayed in Figure 7, show that the additional latency for data sizes below 100 kB does not slow down Redis operations enough for ABDB to catch up with them. ABDB's ability to relocate data shows a more prominent effect above 100 kB of data size, from which it starts to perform significantly better than Redis for both operations.

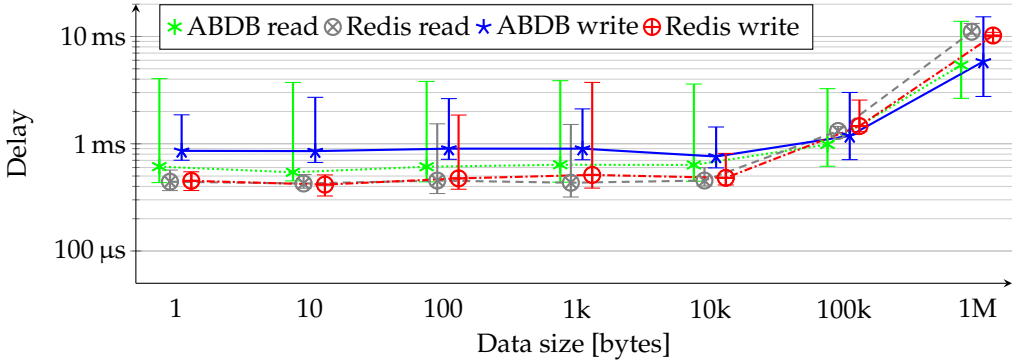

Figure 7. Data access latency of functions deployed to an edge node where only an ABDB child is available. Redis can be accessed on a separate node with a delay of approximately 0.155 ms. Maximum, average and minimum values are shown.

Additional network delay can further worsen Redis's performance. Repeating the same test with an additional delay of 5 ms highlights the potential of the data relocation mechanism of ABDB. As Figure 8 clearly shows, average ABDB access delays are much lower than those of Redis. For data sizes under 10 kB, the average difference is around 4 ms, as can be expected from the configured delay, while, at 100 kB, it is 17–19 ms, and for 1 MB, it is 20–34 ms, depending on the operation type, i.e., read or write.

We note that Redis can run in multi-node setups as well, however, it has a dedicated instance where all write operations have to be forwarded. Thus our observations apply to cases when functions perform write operations on nodes other than the Redis master. Although replicated data can be read from other nodes in the Redis cluster, replication would still suffer from network latency and reading a replica might return stale data.

In conclusion, we can say that in cases when a single function accesses a key and during the lifetime of the application when the function is relocated in a setup with multiple edge nodes, ABDB is a better choice for our platform than Redis when data sizes are above 100 kB or the round-trip time between edge nodes is above 1.5 ms. The observation can be

extended to cases when multiple functions access the same keys but are relocated together. In the following section, we investigate this aspect.

Figure 8. Data access performance of functions deployed to an edge node where only an ABDB child is available. Redis can be accessed on a separate node with a configured delay offset of 5 ms. Maximum, average and minimum values are shown.

5.2. Placement Dependency of Data Access Delay

In the next tests, we utilize a slightly modified edge network setup, shown in Figure 9, where we introduce a Redis instance deployed to the AWS cloud and slightly modified network round-trip times are assumed according to Table 1. The network delay between *Edge node 1* and the cloud is the actual round-trip time between our premises and the closest AWS region, located in Frankfurt. For the rest of the connections, we emulate locations that are slightly farther away from the data center. For *Cloud Redis*, we deploy a *t2.micro* Amazon EC2 instance, with Redis installed with default settings and password authentication in the same way as in our edge setup. This setup is used as an alternative to ElastiCache for Redis in the cloud, as its instances can only be accessed from within an Amazon VPC (Virtual Private Cloud, logically isolated network within an AWS region), and Greengrass functions on the edge cannot access these by default.

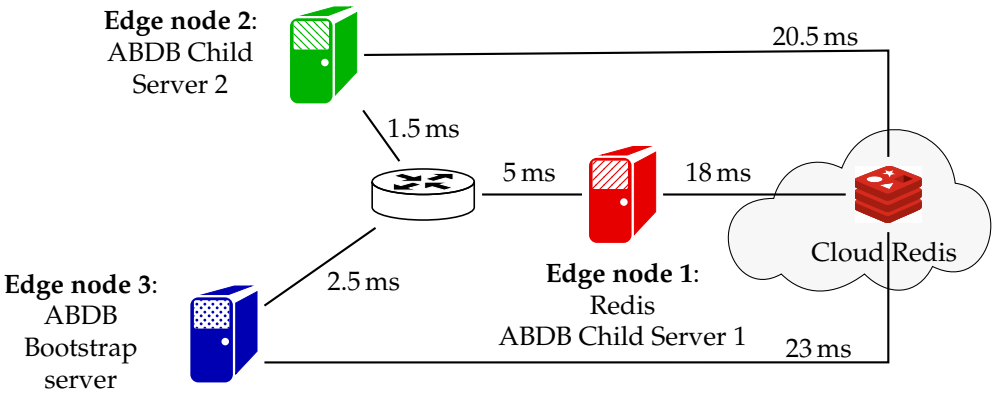

Figure 9. Experimental edge and network setup with three edge nodes and connection to the cloud.

Table 1. Round-trip times between endpoints in the test setup.

	Edge Node 1	Edge Node 2	Edge Node 3	Cloud
Edge node 1	<0.1 ms	<6.5 ms	<7.5 ms	<18 ms
Edge node 2	<6.5 ms	<0.1 ms	<4 ms	<20.5 ms
Edge node 3	<7.5 ms	<4 ms	<0.1 ms	<23 ms
Cloud	18 ms	20.5 ms	23 ms	N/A

In this setup, we deploy an application consisting of 21 functions, with which we emulate complex applications. All application functions are mapped to standalone Lambda functions in this case to achieve the widest range of function mobility and placement options. We have a dedicated *Controller* function that invokes all other functions once in every minute. The rest of the functions run for one minute and they perform read and write operations on the same key according to the intensities specified in Table 2 in every second. If data store access delay prevents a function from reaching the specified number of operations, these are skipped and will not be retried. In a real-life application, this would manifest as increased end-to-end latency and an increased number of parallel function instances. The dedicated *Controller* function is always deployed to *Edge node 1*, whereas data store access test functions f_1–f_{20} are deployed to all edge nodes according to predefined schemes. All function instances on all edge nodes are warmed up before the test starts.

Table 2. Read and write intensities of functions in the experimental setup.

Function	Read [1/s]	Write [1/s]	Function	Read [1/s]	Write [1/s]
f_1	6	8	f_{11}	3	2
f_2	13	8	f_{12}	12	10
f_3	8	10	f_{13}	8	10
f_4	14	7	f_{14}	13	10
f_5	6	11	f_{15}	5	5
f_6	10	7	f_{16}	6	5
f_7	8	11	f_{17}	10	8
f_8	6	13	f_{18}	7	12
f_9	11	8	f_{19}	10	5
f_{10}	7	13	f_{20}	9	11

5.2.1. Effects of Function Relocation

With this iteration of tests, we set ABDB's *Access Pattern Monitor* and *Replacement Initiator* so that access pattern changes are searched within a 20 s time window. Figure 10 shows the test results when we deploy our benchmark application and move functions among edge nodes every 2 min, leveraging the functionalities provided by the Offload Controller component of our platform. The function to node assignment scheme can be seen in the bottom part of the figure, while access delay changes can be observed in the top part. As is visible in the figure, access pattern changes are usually handled by ABDB within the allotted time window of 20 s (see the 4 and 8 min marks). At the startup of the application, this process is even faster (see the 0 min mark) but occasionally slower (minutes 2 and 6). Comparing this to Redis' performance, we can observe that it has much less variation in access delay as data are never relocated. In the beginning, most functions are assigned to *Edge node 2*; this makes it possible for ABDB to calculate an optimal assignment for the single key to this edge node, and move the key there. This also means that accessing data in the Redis instance, located on *Edge node 1*, obtains a significant penalty and the average difference between the two data stores' access delay can reach 4.5 ms in our test setup. This is smaller than the maximum possible delay of 6.5 ms (the round-trip time between *Edge node 1* and *2*) as function f_{20} is assigned to the node on which Redis is available and ABDB's access is generally slower than Redis'. As we move forward in time, function assignment between edge nodes becomes more balanced. In the sixth iteration (minutes 10–12), the assignment is the most balanced and ABDB's and Redis' performance are the closest. In the last iteration, most functions are moved to *Edge node 1*, which favors Redis; however, the few functions left on other nodes modify read delays just enough so that ABDB is almost on par with Redis in this aspect. As access delays for the Redis instance located in the cloud domain show, the best performance is reached at 19 ms, while, in most cases, the access delay is around 20 ms, which can be a deal-breaker for most latency-sensitive applications.

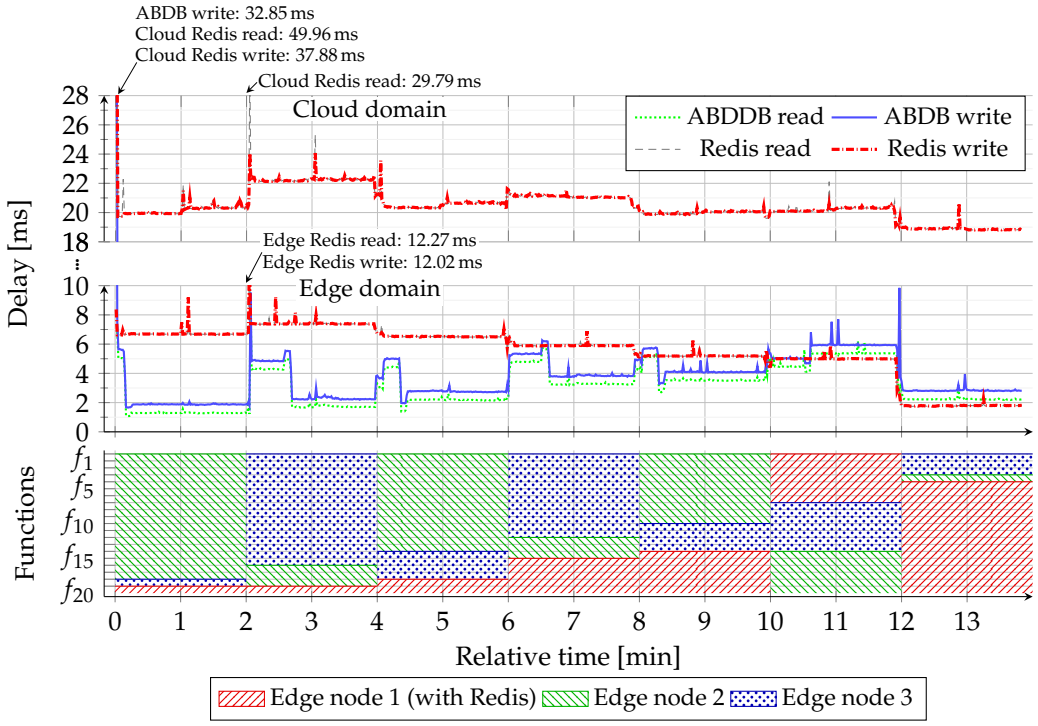

Figure 10. Comparison of ABDB and Redis access performance (**top**) while relocating functions among nodes according to a predefined scheme (**bottom**).

We can summarize our observations in the following way: ABDB receives a performance boost whenever functions are not located strictly on the same node as Redis. In the following section, we explore this aspect on aggregated results gained from multiple deployment scenarios.

5.2.2. Aggregated Results

In the following series of tests, we execute 62 different test cases altogether. Half of them contain deployment schemes that favor Redis, i.e., functions gravitate towards *Edge node 1*. The other half of the cases consist of schemes where functions are deployed to *Edge node 2* or *3*. These cases cover all configurations made possible by the three physical edge nodes at our disposal. We categorize the measurement results of the cases based on the dispersion of functions between the edge nodes. As the measure of dispersion, we use the standard deviation of the number of functions on each edge node, i.e., in the most balanced case, where 7 of the total 20 test functions are assigned to *Edge node 1* and *Edge node 2*, respectively, and the remaining 6 to *Edge node 3*, we obtain a dispersion value of 0.47. In the most unbalanced case, i.e., when every function is assigned to the same node (to *Edge node 1* in this case), the metric is 9.43. In order to analyze trends in the results, we group the cases into 10 bins; thus, the first case in our example above would fall into the first bin and the other into the last. We use the balanced case given in the example, and select a less extreme case for the other end of the spectrum, where all edge nodes run at least one function. During these simple tests, we utilize the same access intensities as shown in Table 2 and run them for 2 min without function relocation. Here, we note that the number of test cases is not mapped uniformly to the dispersion intervals, as depicted in Figure 11, and thus our interpretation inherently manifests as higher variance in the access delay in the midsection of the figures showing the results.

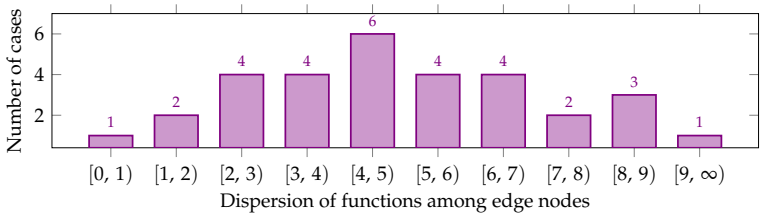

Figure 11. Number of cases in each dispersion interval.

Figure 12 shows aggregated results gained from running the cases where Redis is favored. The figure's top half shows read, the lower part write performance. Looking at the figure, we can observe that, in a balanced case (i.e., in the $[0, 1)$ dispersion interval), ABDB and Redis are almost on par with each other. Here, ABDB's read delay is slightly lower than that of Redis (the exact difference being 351 µs), while its write delay is 218 µs higher than Redis'. As can be expected, this changes quickly as the function placement starts to gravitate more towards *Edge node 1* until we reach a difference of 1.27 ms in read delay and 1.87 ms in write delay at the highest dispersion.

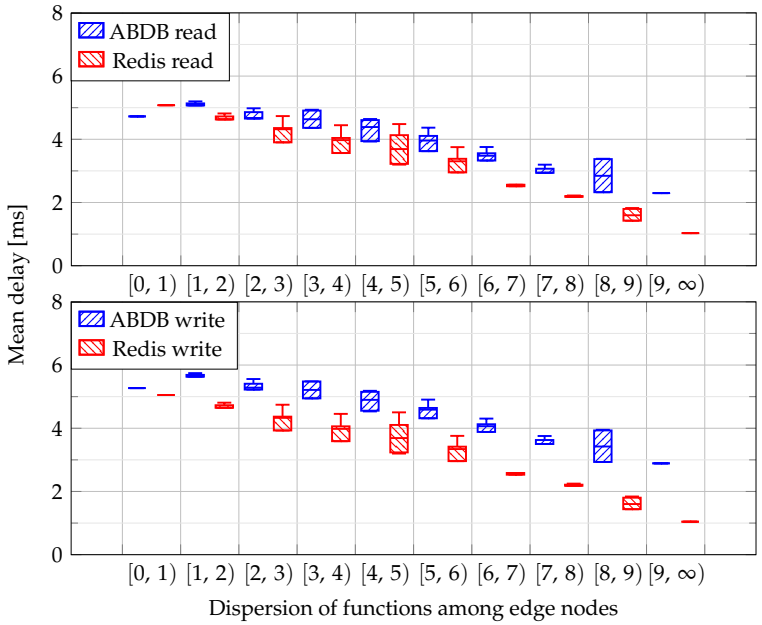

Figure 12. Comparison of ABDB and Redis read and write delays in the "Redis-favored" scenario.

The second setup disfavors Redis by mostly avoiding *Edge node 1* for function placement. Figure 13 shows that, in such cases, ADBD always performs better in both read and write operations. Here, ABDB provides lower read and write delays even in the balanced case, with the read difference being 847 µs and write difference 280 µs. We note that this difference is attributed to the fact that, using our 20 functions, we cannot create a completely balanced case, and, in the previous, "Redis-favored" scenario, we placed 7 functions on the Redis node, while, in this "Redis-disfavored" case, only 6 of them are placed on the node. When the performance gap is the widest, ABDB can read data 4.88 ms faster than Redis and write 4.33 ms quicker on average.

It might happen that, due to a specific dispersion of the functions, the experienced mean delay significantly differs from the other measured values of the same bin. We depict these deviations as outliers in the figures. As an example, we examine the case where nine functions (f_2–f_{11}) run on *Edge node 3*, ten of them (f_{12}–f_{20}) run on *Edge node* 2 and only one (f_1) is deployed on *Edge node 1*. The standard deviation of the number of functions on the nodes is 4.93, which means that the case goes to the bin $[4, 5)$. The ABDB will optimize the location of the data and move them to *Edge node 3*, resulting in a 4 ms access delay for

f_2–f_{11} and 7.5 ms for f_1. In this case, the mean overall delay of all the functions (f_1–f_{20}) is around 2 ms, which is an outlier compared to the other measured values (around 4 ms) from this bin.

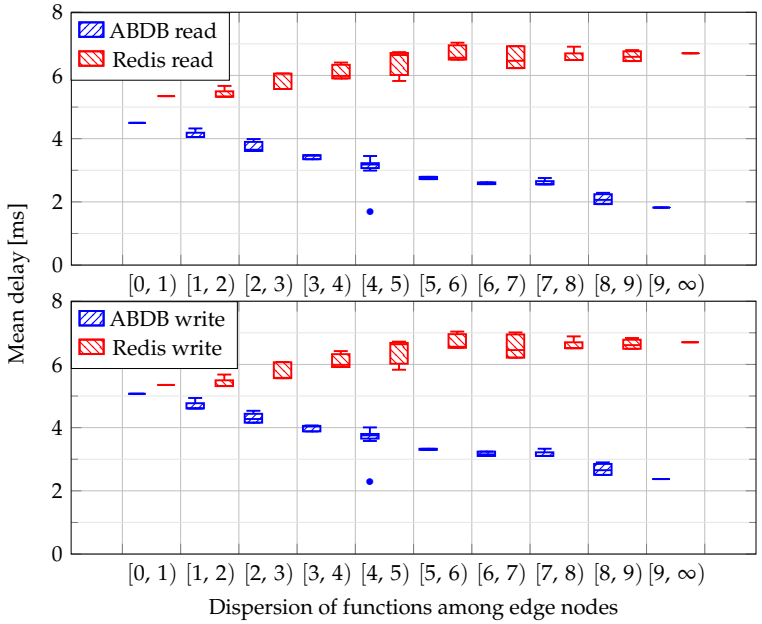

Figure 13. Comparison of ABDB and Redis read and write delays in the "Redis-disfavored" scenario.

To summarize the results, on average, Redis can be a good solution only if functions are located on the edge node where it is running. Therefore, in use-cases where the users of the edge application are moving across edge sites, and the ephemeral functions of the application necessitate externalized data to travel with them, an adaptive location-aware data store is needed, such as ABDB. ABDB is on par with Redis even in local data access scenarios, but, particularly in cases when Redis is obliged to access remote data, ABDB relocates data based on the observed access pattern instead, and thus performs significantly better in terms of latency.

5.2.3. Data Access Simulation of a Complex Automotive Application

Thus far, basic atomic operations such as data read and write have been examined in terms of the data access latency within different data store scenarios. However, most autonomous driving tasks are more complicated, e.g., as Yi et al. argue [60], in the automotive industry, control applications consist of multiple independent tasks, commonly modeled by Directed Acyclic Graphs (DAGs), which represent periodic tasks and their read–write dependencies. The authors present a reference autonomous driving application from Bosch.

In our previous works [8,61], we conducted experiments with a sample application performing image preprocessing and two-stage object detection. The application acquires a video feed showing a traffic scenario with varying numbers of vehicles. During preprocessing, the image size is reduced and the first stage of object detection determines bounding boxes for detected vehicles on the scaled-down images. At a later stage, the original high-resolution image is split according to the output of the first-stage detection, and the second stage of object detection is executed on the cropped images to gather more details about the objects. The application is deployed by our deployment engine based on the layout computed by our Layout and Placement Optimizer, and we examine application performance during component relocation to the cloud utilizing a single Redis instance deployed on the single edge node as a data store.

Another example use-case from the automotive industry appears in [62], in which the authors introduce an end-to-end autonomous driving application, i.e., a reference

application that provides proper throttle, steering and brake signals to drive a vehicle through a predetermined map of way-points. They identify the individual application tasks and the necessary data exchanges between them, as depicted in Figure 14. All in all, nine tasks read and write 16 pieces of data in a period. The authors assume that each task is independently activated once within a period, i.e., they could read or write a single piece of data into the storage once per period, which gives 16 writes and 21 reads per period as the access pattern of this reference application. The access pattern shows which tasks write (blue arrows originating from the compute layer) and read (green arrows originating from the data layer) which data during a period. The authors further elaborate on how the application's tasks operate.

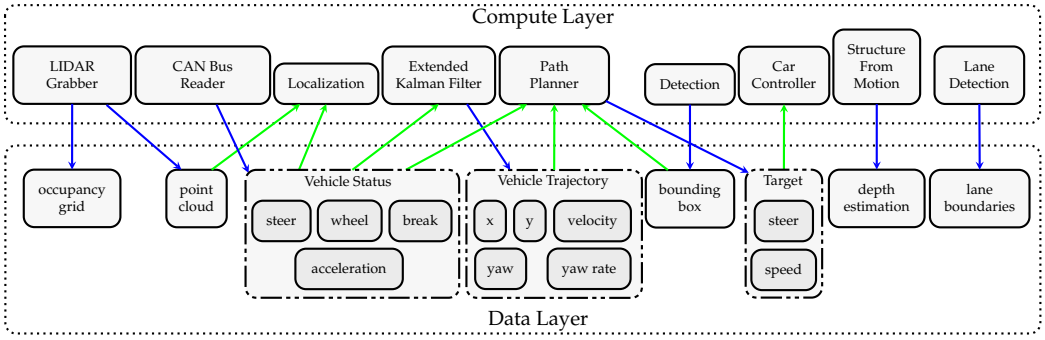

Figure 14. Access pattern of the examined autonomous driving application [62].

One might notice that the access pattern, or, more precisely, the access delay of data, has a significant effect on the end-to-end execution time of the entire application. Consequently, an access pattern-aware data store should be used in such latency-sensitive cases as the presented autonomous driving application. Let us assume a vehicle that is driven by the above-mentioned self-driving application that uses the network setup of Figure 9. As the self-driving application has a strict end-to-end execution latency requirement, all the individual tasks should run on the nearest edge node. The question is where to store the application's internal data. If the vehicle is close to *Edge node 2*, the ABDB will detect it and minimize the summarized access latency by migrating all the 16 data to this node. Another option is to use Redis either on *Edge node 1* or in the *cloud*. Assuming that all the data in the access pattern are 100 kB, Figure 15 depicts the average time that the application must spend to read and write data per period in different data store scenarios.

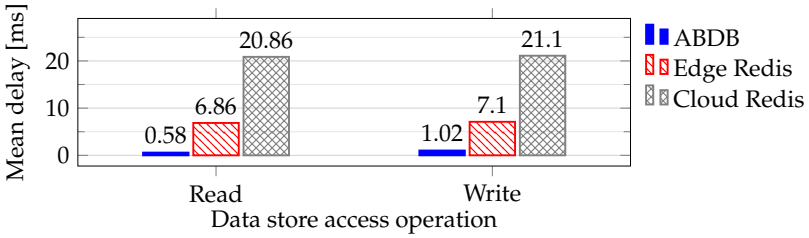

Figure 15. Data read and write delay of the autonomous driving application in case of different data store options.

As ABDB is the only storage option where data placement is optimized based on locality, it achieves the lowest latency both in terms of reading and writing data per period. Therefore, in the case of ABDB, all data will be eventually co-located with the accessing tasks on *Edge node 2*. The farther the data are located from the vehicle's application tasks, the higher is the data access latency that the application has to suffer during a period, which is clearly shown by the increased access delay in the case of the single Redis edge or cloud instance.

6. Discussion

With the emergence of the enabling technology, novel latency-critical applications can now be deployed on edge or fog resources, offloading energy-consuming tasks from end devices. Such an application domain is automated and assisted driving, in which it is essential to run computing functions geographically close to the end-user vehicles in order to ensure that end-to-end latency requirements are not compromised. Besides the proximity, though, the edge computing platform must provide the necessary operation techniques in order to avoid added delays by all means. In this paper, we propose an edge computing platform that comprises orchestration methods with such objectives, in terms of handling the deployment of both functions and data. We build the platform on prior research results and we show how the integration of the function orchestration solution with the adaptive data placement of a distributed key–value store can lead to decreased end-to-end latency even when the mobility of end devices creates a dynamic set of requirements. In the integrated framework, the computing optimization is responsible for the cost-effective and delay-aware composition of the serverless applications' components, while the adaptive placement of data entries within the distributed key–value store server instances closely follows the access patterns of these components towards their externalized data. Along with the necessary monitoring features, the proposed edge platform is capable of serving the nomad users of novel applications with low latency requirements. We showcase this capability in several scenarios, in which we articulate the end-to-end latency performance of our platform by comparing delay measurements with the benchmark of a Redis-based setup lacking the adaptive nature of data orchestration. Our results prove that the stringent delay requisites necessitate the close integration that we present in this paper: functions and data must be orchestrated in sync in order to fully exploit the potential offered by fog or edge environments.

Author Contributions: Conceptualization, J.C., I.P., B.S., M.S. and L.T.; methodology, J.C., I.P., B.S., M.S. and L.T.; software, J.C., I.P. and M.S.; validation, I.P. and M.S.; formal analysis, J.C. and M.S.; investigation, I.P. and M.S.; resources, B.S. and L.T.; data curation, J.C., I.P. and M.S.; writing—original draft preparation, J.C., I.P., B.S., M.S. and L.T.; writing—review and editing, J.C., I.P., B.S., M.S. and L.T.; visualization, I.P. and M.S.; supervision, B.S. and L.T.; project administration, L.T.; funding acquisition, B.S. and L.T. All authors have read and agreed to the published version of the manuscript.

Funding: This work was supported by the Ministry of Innovation and Technology of Hungary from the National Research, Development and Innovation Fund through projects (i) no. 135074 under the FK_20 funding scheme, (ii) 2019-2.1.13-TÉT_IN-2020-00021 under the 2019-2.1.13-TÉT-IN funding scheme, and (iii) 2018-2.1.17-TÉT-KR-2018-00012 under the 2018-2.1.17-TÉT-KR funding scheme. L. Toka was supported by the János Bolyai Research Scholarship of the Hungarian Academy of Sciences. Supported by the ÚNKP-21-5 New National Excellence Program of the Ministry for Innovation and Technology from the source of the National Research, Development and Innovation Fund.

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Data Availability Statement: Not applicable.

Acknowledgments: This work has been accomplished in cooperation with HSN Labor Kft., which provided expertise in edge/cloud applications and data store operation, and access to AWS resources.

Conflicts of Interest: The authors declare no conflicts of interest.

Abbreviations

The following abbreviations are used in this manuscript:

ABDB	AnnaBellaDB
API	Application Programming Interface
AWS	Amazon Web Services
CLI	Command Line Interface
CPU	Central Processing Unit
DAG	Directed Acyclic Graph
EC2	Amazon Elastic Compute Cloud
FaaS	Function as a Service
IoT	Internet of Things
KVS	Key–Value Store
LPO	Layout and Placement Optimizer
MEC	Mobile Edge Computing
NFV	Network Function Virtualization
OC	Offload Controller
RAN	Radio Access Network
SDE	Serverless Deployment Engine
SDK	Software Development Kit
SDN	Software-Defined Network
SFC	Service Function Chaining
vCPU	Virtual Central Processing Unit
VM	Virtual Machine
VNF	Virtual Network Function
YAML	Yet Another Markup Language

References

1. Gomes, E.; Costa, F.; De Rolt, C.; Plentz, P.; Dantas, M. A Survey from Real-Time to Near Real-Time Applications in Fog Computing Environments. *Telecom* **2021**, *2*, 489–517. doi:10.3390/telecom2040028.
2. Szalay, M.; Nagy, M.; Géhberger, D.; Kiss, Z.; Mátray, P.; Németh, F.; Pongrácz, G.; Rétvári, G.; Toka, L. Industrial-scale stateless network functions. In Proceedings of the 2019 IEEE 12th International Conference on Cloud Computing (CLOUD), Milan, Italy, 8–13 July 2019; pp. 383–390.
3. Amazon Web Services Inc. Amazon Web Services. 2021. Available online: https://aws.amazon.com (accessed on 28 November 2021).
4. Google Cloud: Google Kubernetes Engine (GKE as Caas) and Google Cloud Functions (FaaS). 2021. Available online: https://cloud.google.com/ (accessed on 30 November 2021).
5. Microsoft Azure: Azure Kubernetes Service (AKS as CaaS) and Azure Functions (FaaS). 2021. Available online: https://azure.microsoft.com/ (accessed on 30 November 2021).
6. Haja, D.; Turanyi, Z.R.; Toka, L. Location, Proximity, Affinity – The key factors in FaaS. *Infocommun. J.* **2020**, *12*, 14–21. doi:10.36244/ICJ.2020.4.3.
7. Szalay, M.; Mátray, P.; Toka, L. State Management for Cloud-Native Applications. *Electronics* **2021**, *10*, 423.
8. Pelle, I.; Czentye, J.; Doka, J.; Kern, A.; Gero, B.P.; Sonkoly, B. Operating Latency Sensitive Applications on Public Serverless Edge Cloud Platforms. *IEEE Internet Things J.* **2021**, *8*, 7954–7972. doi:10.1109/jiot.2020.3042428.
9. Szalay, M.; Matray, P.; Toka, L. AnnaBellaDB: Key-Value Store Made Cloud Native. In Proceedings of the 2020 16th International Conference on Network and Service Management (CNSM), IEEE, Izmir, Turkey, 2–6 November 2020; pp. 1–5.
10. Da Silva, M.D.; Tavares, H.L. *Redis Essentials*; Packt Publishing Ltd.: Birmingham, UK, 2015.
11. Yang, F.; Wang, S.; Li, J.; Liu, Z.; Sun, Q. An overview of Internet of Vehicles. *China Commun.* **2014**, *11*, 1–15. doi:10.1109/CC.2014.6969789.
12. Contreras-Castillo, J.; Zeadally, S.; Guerrero-Ibañez, J.A. Internet of Vehicles: Architecture, Protocols, and Security. *IEEE Internet Things J.* **2018**, *5*, 3701–3709. doi:10.1109/JIOT.2017.2690902.
13. Continental AG. Continental Continues to Drive Forward the Development of Server-Based Vehicle Architectures. 2021. Available online: https://www.continental.com/en/press/press-releases/20210728-cross-domain-hpc/ (accessed on 30 November 2021).
14. Continental AG. Continental and Amazon Web Services Create Platform for Automotive Software. 2021. Available online: https://www.continental.com/en/20210415-continental-and-amazon-web-services/ (accessed on 30 November 2021).
15. Rahimi, M.R.; Venkatasubramanian, N.; Mehrotra, S.; Vasilakos, A.V. On Optimal and Fair Service Allocation in Mobile Cloud Computing. *IEEE Trans. Cloud Comput.* **2018**, *6*, 815–828. doi:10.1109/tcc.2015.2511729.

16. Zakarya, M.; Gillam, L.; Ali, H.; Rahman, I.; Salah, K.; Khan, R.; Rana, O.; Buyya, R. epcAware: A Game-based, Energy, Performance and Cost Efficient Resource Management Technique for Multi-access Edge Computing. *IEEE Trans. Serv. Comput.* **2020**, 1. doi:10.1109/tsc.2020.3005347.
17. Chantre, H.D.; da Fonseca, N.L. Multi-objective optimization for edge device placement and reliable broadcasting in 5G NFV-based small cell networks. *IEEE J. Sel. Areas Commun.* **2018**, *36*, 2304–2317.
18. Mouradian, C.; Kianpisheh, S.; Abu-Lebdeh, M.; Ebrahimnezhad, F.; Jahromi, N.T.; Glitho, R.H. Application component placement in NFV-based hybrid cloud/fog systems with mobile fog nodes. *IEEE J. Sel. Areas Commun.* **2019**, *37*, 1130–1143.
19. Yang, L.; Cao, J.; Liang, G.; Han, X. Cost aware service placement and load dispatching in mobile cloud systems. *IEEE Trans. Comput.* **2015**, *65*, 1440–1452.
20. Ceselli, A.; Premoli, M.; Secci, S. Mobile edge cloud network design optimization. *IEEE/ACM Trans. Netw.* **2017**, *25*, 1818–1831.
21. Yang, B.; Chai, W.K.; Pavlou, G.; Katsaros, K.V. Seamless support of low latency mobile applications with nfv-enabled mobile edge-cloud. In Proceedings of the 2016 5th IEEE International Conference on Cloud Networking (Cloudnet), Pisa, Italy, 3–5 October 2016; pp. 136–141.
22. Badri, H.; Bahreini, T.; Grosu, D.; Yang, K. Energy-aware application placement in mobile edge computing: A stochastic optimization approach. *IEEE Trans. Parallel Distrib. Syst.* **2019**, *31*, 909–922.
23. Ochoa-Aday, L.; Cervelló-Pastor, C.; Fernández-Fernández, A.; Grosso, P. An Online Algorithm for Dynamic NFV Placement in Cloud-Based Autonomous Response Networks. *Symmetry* **2018**, *10*, 163. doi:10.3390/sym10050163.
24. Herrera, J.G.; Botero, J.F. Resource Allocation in NFV: A Comprehensive Survey. *IEEE Trans. Netw. Serv. Manag.* **2016**, *13*, 518–532. doi:10.1109/tnsm.2016.2598420.
25. Bhamare, D.; Jain, R.; Samaka, M.; Erbad, A. A survey on service function chaining. *J. Netw. Comput. Appl.* **2016**, *75*, 138–155. doi:10.1016/j.jnca.2016.09.001.
26. Abdelaal, M.A.; Ebrahim, G.A.; Anis, W.R. Efficient Placement of Service Function Chains in Cloud Computing Environments. *Electronics* **2021**, *10*, 323. doi:10.3390/electronics10030323.
27. Wu, Y.; Zhou, J. Dynamic Service Function Chaining Orchestration in a Multi-Domain: A Heuristic Approach Based on SRv6. *Sensors* **2021**, *21*, 6563. doi:10.3390/s21196563.
28. Santos, G.L.; de Freitas Bezerra, D.; da Silva Rocha, É.; Ferreira, L.; Moreira, A.L.C.; Gonçalves, G.E.; Marquezini, M.V.; Recse, Á.; Mehta, A.; Kelner, J.; et al. Service Function Chain Placement in Distributed Scenarios: A Systematic Review. *J. Netw. Syst. Manag.* **2021**, *30*, 1–39. doi:10.1007/s10922-021-09626-4.
29. Sonkoly, B.; Czentye, J.; Szalay, M.; Nemeth, B.; Toka, L. Survey on Placement Methods in the Edge and Beyond. *IEEE Commun. Surv. Tutor.* **2021**, *23*, 2590–2629. doi:10.1109/comst.2021.3101460.
30. Baldini, I.; Castro, P.; Chang, K.; Cheng, P.; Fink, S.; Ishakian, V.; Mitchell, N.; Muthusamy, V.; Rabbah, R.; Slominski, A.; et al. Serverless Computing: Current Trends and Open Problems. In *Research Advances in Cloud Computing*; Springer: Singapore, 2017; pp. 1–20. doi:10.1007/978-981-10-5026-8_1.
31. Kjorveziroski, V.; Filiposka, S.; Trajkovik, V. IoT Serverless Computing at the Edge: A Systematic Mapping Review. *Computers* **2021**, *10*, 130. doi:10.3390/computers10100130.
32. Dormando. Memcached-A Distributed Memory Object Caching System. 2018. Available online: https://memcached.org/ (accessed on 28 November 2021).
33. Lakshman, A.; Malik, P. Cassandra: A decentralized structured storage system. *ACM SIGOPS Oper. Syst. Rev.* **2010**, *44*, 35–40.
34. Sivasubramanian, S. Amazon dynamoDB: A seamlessly scalable non-relational database service. In Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data, Scottsdale, AZ, USA, 20–24 May 2012; pp. 729–730.
35. Wu, C.; Sreekanti, V.; Hellerstein, J.M. Autoscaling tiered cloud storage in Anna. *Proc. VLDB Endow.* **2019**, *12*, 624–638. doi:10.14778/3311880.3311881.
36. Serverless Inc. The Serverless Application Framework. 2021. Available online: https://serverless.com/ (accessed on 28 Novermber 2021).
37. HashiCorp. Terraform by HashiCorp. 2021. Available online: https://www.terraform.io (accessed on 28 Novermber 2021).
38. Cirba Inc. Densify: Hybrid Cloud & Container Resource Management & Optimization. 2021. Available online: https://www.densify.com (accessed on 28 Novermber 2021).
39. Amazon Web Services Inc. AWS CloudFormation. 2021. Available online: https://aws.amazon.com/cloudformation/ (accessed on 28 Novermber 2021).
40. Stackery. Stackery. 2021. Available online: https://www.stackery.io (accessed on 28 Novermber 2021).
41. Amazon Web Services Inc. AWS Compute Optimizer. 2021. Available online: https://aws.amazon.com/compute-optimizer/ (accessed on 28 Novermber 2021).
42. Eismann, S.; Grohmann, J.; van Eyk, E.; Herbst, N.; Kounev, S. Predicting the Costs of Serverless Workflows. In Proceedings of the ACM/SPEC International Conference on Performance Engineering (ICPE), Edmonton, AB, Canada, 20–24 April 2020; pp. 265–276.
43. Fotouhi, M.; Chen, D.; Lloyd, W.J. Function-as-a-Service Application Service Composition: Implications for a Natural Language Processing Application. In *Proceedings of the 5th International Workshop on Serverless Computing (WOSC)*; ACM: New York, NY, USA, 2019; pp. 49–54.

44. Winzinger, S.; Wirtz, G. Model-Based Analysis of Serverless Applications. In *Proceedings of the 11th International Workshop on Modelling in Software Engineerings (MiSE)*; ACM: New York, NY, USA, 2019; pp. 82–88.
45. Kuhlenkamp, J.; Klems, M. Costradamus: A Cost-Tracing System for Cloud-Based Software Services. In *Service-Oriented Computing*; Springer: Berlin, Germany, 2017; pp. 657–672.
46. Mahajan, K.; Figueiredo, D.; Misra, V.; Rubenstein, D. Optimal Pricing for Serverless Computing. In Proceedings of the 2019 IEEE Global Communications Conference (GLOBECOM), Waikoloa, HI, USA, 9–13 December 2019; pp. 1–6.
47. Elgamal, T. Costless: Optimizing Cost of Serverless Computing through Function Fusion and Placement. In Proceedings of the 2018 IEEE/ACM Symposium on Edge Computing (SEC), Seattle, WA, USA, 25–27 October 2018; pp. 300–312.
48. Das, A.; Imai, S.; Wittie, M.P.; Patterson, S. Performance Optimization for Edge-Cloud Serverless Platforms via Dynamic Task Placement. In Proceedings of the 2020 20th IEEE/ACM International Symposium on Cluster, Cloud and Internet Computing (CCGRID), Melbourne, VIC, Australia, 11–14 May 2020.
49. Mahmoudi, N.; Lin, C.; Khazaei, H.; Litoiu, M. Optimizing Serverless Computing: Introducing an Adaptive Function Placement Algorithm. In *Proceedings of the 29th Annual International Conference on Computer Science and Software Engineering (CASCON)*; ACM: New York, NY, USA, 2019; pp. 203–213.
50. Bravetti, M.; Giallorenzo, S.; Mauro, J.; Talevi, I.; Zavattaro, G. Optimal and Automated Deployment for Microservices. In *Fundamental Approaches to Software Engineering*; Springer: Berlin, Germany, 2019; pp. 351–368.
51. Pelle, I.; Paolucci, F.; Sonkoly, B.; Cugini, F. Fast Edge-to-Edge Serverless Migration in 5G Programmable Packet-Optical Networks. In *Optical Fiber Communication Conference (OFC) 2021*; Optical Society of America: Washington, DC, USA, 2021; p. W1E.1. doi:10.1364/OFC.2021.W1E.1.
52. Amazon Web Services Inc. AWS Lambda: Serverless Computing. 2021. Available online: https://aws.amazon.com/lambda/ (accessed on 28 November 2021).
53. Amazon Web Services Inc. Intelligence at the IoT Edge–AWS IoT Greengrass. 2021. Available online: https://aws.amazon.com/greengrass/ (accessed on 28 November 2021).
54. Pelle, I.; Czentye, J.; Dóka, J.; Sonkoly, B. Towards Latency Sensitive Cloud Native Applications: A Performance Study on AWS. In Proceedings of the 2019 IEEE 12th International Conference on Cloud Computing (CLOUD), Milan, Italy, 8–13 July 2019; pp. 272–280. doi:10.1109/CLOUD.2019.00054.
55. Czentye, J.; Pelle, I.; Kern, A.; Gero, B.P.; Toka, L.; Sonkoly, B. Optimizing Latency Sensitive Applications for Amazon's Public Cloud Platform. In Proceedings of the 2019 IEEE Global Communications Conference (GLOBECOM), Waikoloa, HI, USA, 9–13 December 2019; pp. 1–7.
56. Baldini, I.; Cheng, P.; Fink, S.J.; Mitchell, N.; Muthusamy, V.; Rabbah, R.; Suter, P.; Tardieu, O. The serverless trilemma: Function composition for serverless computing. In *Proceedings of the 2017 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*; ACM: New York, NY, USA, 2017. doi:10.1145/3133850.3133855.
57. Skarin, P.; Tarneberg, W.; Arzen, K.E.; Kihl, M. Control-over-the-cloud: A performance study for cloud-native, critical control systems. In Proceedings of the 2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC), Leicester, UK, 7–10 December 2020. doi:10.1109/ucc48980.2020.00025.
58. Amazon EC2 Spot Instances. Available online: https://aws.amazon.com/ec2/spot/ (accessed on 28 November 2021).
59. Szalay, M.; Mátray, P.; Toka, L. Minimizing state access delay for cloud-native network functions. In Proceedings of the 2019 IEEE 8th International Conference on Cloud Networking (CloudNet), Coimbra, Portugal, 4–6 November 2019; pp. 1–6.
60. Yi, S.; Kim, T.W.; Kim, J.C.; Dutt, N. Energy-Efficient adaptive system reconfiguration for dynamic deadlines in autonomous driving. In Proceedings of the 2021 IEEE 24th International Symposium on Real-Time Distributed Computing (ISORC), Daegu, Korea, 1–3 June 2021; pp. 96–104.
61. Pelle, I.; Czentye, J.; Dóka, J.; Sonkoly, B. Dynamic Latency Control of Serverless Applications Operated on AWS Lambda and Greengrass. In *Proceedings of the SIGCOMM '20 Poster and Demo Sessions; Association for Computing Machinery*; SIGCOMM '20; ACM: New York, NY, USA, 2020; pp. 33–34. doi:10.1145/3405837.3411381.
62. Wurst, F.; Dasari, D.; Hamann, A.; Ziegenbein, D.; Sanudo, I.; Capodieci, N.; Bertogna, M.; Burgio, P. System performance modelling of heterogeneous hw platforms: An automated driving case study. In Proceedings of the 2019 22nd Euromicro Conference on Digital System Design (DSD), Kallithea, Greece, 28–30 August 2019; pp. 365–372.