Edge Offloading for Microservice Architectures

Josip Zilic
Vienna University of Technology
josip.zilic@tuwien.ac.at

Vincenzo De Maio
Vienna University of Technology
vincenzo@ec.tuwien.ac.at

Atakan Aral
University of Vienna
atakan.aral@univie.ac.at

Ivona Brandic
Vienna University of Technology
ivona.brandic@tuwien.ac.at

ABSTRACT

Edge offloading is widely used to support the execution of near real-time mobile applications. However, offloading on edge infrastructures can suffer from failures due to the absence of supporting systems and environmental factors. We propose a fault-tolerant offloading method modeled as a Markov Decision Process (MDP) based on predictions performed through Support Vector Regression (SVR). SVR is used to estimate offloading service availability, which is used by MDP for offloading decisions. Our approach is implemented in a real-world test-bed and compared with the default Kubernetes scheduler augmented with hybrid fault-tolerance.

CCS CONCEPTS

• **Computer systems organization → Distributed architectures**; • **Computing methodologies → Distributed computing methodologies**.

KEYWORDS

edge offloading; kubernetes; containers; microservices

1 INTRODUCTION

Offloading applications (or parts of them) on remote surrogate machines can reduce resource consumption of mobile devices [21]. While offloading delay-tolerant applications on far distanced cloud servers can increase energy efficiency, (near-)real-time mobile applications (e.g., augmented virtual reality, live traffic navigation) requires Edge offloading [14], i.e., offloading to nearby edge devices to address latency constraints [30]. This approach enables running new emerging consumer-oriented offloading applications. For instance, web browser can be accelerated by offloading browsing functions (e.g. content caching, optimizing transmission) on remote edge nodes [24]. It alleviates backhaul network traffic and can lead to radio network optimization based on real-time and runtime information for improving network and QoE quality levels [14]. Previous works [9, 12] provide insights that coupling together edge computing platform, microservice architecture, and container orchestration can provide a modular and loosely-coupled edge architecture to address the resource limitations. However, the absence of supporting systems on the edge devices (e.g., cooling) and environmental factors can cause failures that affect offloading [3].

Studies in [10, 18, 28] target edge offloading on a failure-free edge IoT-enabled infrastructure and stateful vehicles without considering proactive fault-tolerant measures and real-world experimentation. Also, research focused on offloading in a failure-prone environment mostly considered reactive fault tolerance, such as check-pointing [16] and local re-computing [27], which can cause high execution delays. Moreover, reactive recovery actions in microservice applications can cause interference to other services [29]. Work [25] shows how proactive fault tolerance can improve cloud servers' performance w.r.t. reactive failure management.

We propose an edge offloading algorithm that employs Markov Decision Process (MDP) which performs proactive fault tolerance based on predictions obtained through Support Vector Regression (SVR). The SVR algorithm predicts offloading service availability on remote sites and forwards those predictions to the MDP-based decision engine on a mobile device that synthesizes the offloading decision policy for task offloading. We select the SVR algorithm due to its prediction accuracy above 90% for failure time-series data [15] and its relatively small training dataset [6] w.r.t. deep neural networks. Also, MDPs allow to model edge offloading due to numerous offloading service alternatives and stochastic availability. Remote offloading services are implemented as micro-services running in Docker containers and deployed on Kubernetes cluster. Exposing them as public Kubernetes

EdgeSys'22, April 5–8, 2022, RENNES, France

ACM ISBN 978-1-4503-9253-2/22/04…$15.00
https://doi.org/10.1145/3517206.3526266

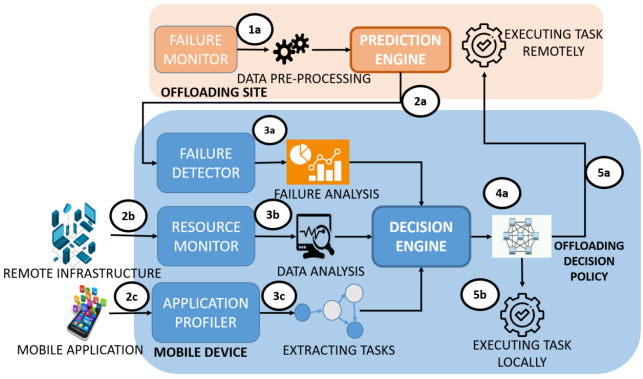


Figure 1: Edge Offloading Framework

services through HTTP APIs, enables them to receive offloaded application tasks from mobile device where decision engine is placed. The offloading framework is evaluated on an experimental test-bed and compared to the baseline Kubernetes scheduler augmented with hybrid fault-tolerance.

Edge offloading is described in Section 2. Then, we describe offloading framework design and offloading algorithm in Section 3. Section 4 describes prototype implementation. In Section 5 we describe evaluation results. Finally, we describe related work in Section 6 and conclude the paper in Section 7.

2 EDGE OFFLOADING

Edge Offloading is the process of executing an application (or part of it) to remote computational nodes, to improve performance like conserving battery energy supplies and reducing application runtime. Offloading requires deciding whether and where to offload a task, depending on task characteristics and network availability [13] and according to different objectives. To address these issues, [7] envisions three main components on mobile device: (*i*) *system monitor*, which collects resource information about the remote infrastructure; (*ii*) *application profiler*, which extracts tasks and resource requirements of mobile applications, and (*iii*) *decision engine*, which takes offloading decisions based on other components' data. However, the impact of network failures is not considered. Thus, we add proactive fault tolerance to [7], as opposed to typical approaches based on reactive fault tolerance [16, 27].

3 SYSTEM DESIGN

3.1 Edge Offloading Framework

We envision an Edge Offloading Framework with the following components: (*i*) *Decision engine*, which computes the offloading decision policy; (*ii*) *Prediction engine*, which estimates future service availability on the remote offloading sites based on local historical failure trace logs; (*iii*) *Failure monitor*, which monitors failures of local system operations on remote offloading sites; (*iv*) *Failure detector*, which detects failures during execution on remote offloading sites and collects the failure estimation data from prediction engine; (*v*) *Resource monitor*, which collects resource information about remote infrastructure; (*vi*) *Application profiler*, which profiles resource requirements of underlying mobile applications. Components are partitioned between mobile device and remote offloading sites as summarized in Figure 1.

The edge offloading process is described as following. First, the failure monitor collects historical failure traces and forwards them to the prediction engine (step 1a), which estimates service availability of each offloading site and sends these data to a mobile device (step 2a). Simultaneously, the application profiler and the resource monitor collect data about mobile application requirements and remote infrastructure capabilities (steps 2b and 2c). These data are used by the decision engine (steps 3a, 3b, and 3c) for offloading decisions (Step 4a), based on which tasks are offloaded (Step 5a and 5b).

3.1.1 Application Requirements. We focus on response time (RT) and mobile device battery lifetime (BL) as in [31]. RT is defined as the sum of local computation time, uploading, and downloading data transfer time. Local computation time is defined as a ratio between CPU Millions of Instructions per Second (MIPS) and the number of task's instructions. Data transfer time is defined as a ratio between data size and network bandwidth plus the network latency.

BL is defined as the difference between total battery capacity and runtime energy consumption. Energy consumption is defined as the sum of local, upload, and download energy consumption. Each energy consumption component is equal to the multiplication of time and its power coefficient. We assume $p_u > p_d > p_c > p_i$, respectively power consumption for upload, download, local computation and idle [19].

3.1.2 Offloading Sites. We assume the infrastructure setup of [31], which allows to address diverse application requirements, i.e., data-intensive, computational-intensive, and moderate applications. We assume three Edge nodes types: (i) *Edge database server* (ED), with large data storage capabilities and fast network transmission rates for data-intensive applications; (ii) *Edge computational server* (EC) with greater computational power to support computational-intensive applications such as games and AI, and (iii) *Edge regular server* (ER) with intermediate resources suitable for applications that do not require a large amount of computation or data storage capabilities, such as live traffic navigation or posting updates on Facebook. Edge nodes are clustered together with the cloud data center (CD).

Algorithm 1 Edge Offloading Algorithm

```
1: procedure EDGE_OFF_ALGO(S, A, train_dataset, tasks)
2:     energy_vector ← array()      ▷ Store energy consumption of each off. site
3:     time_vector ← array()        ▷ Store response time for each offloading site
4:     for each state v in tasks do
5:         for each state q in S do
6:             energy ← compute_energy(v, q)
7:             time ← compute_time(v, q)
8:             energy_vector.append(energy)
9:             time_vector.append(time)
10:        end for
11:    end for
12:    svr_avail_predict ← SVR(train_dataset)          ▷ Predict availability
13:    P ← compute_P_matrix(svr_avail_predict)
14:    R ← compute_R_matrix(energy_vector, time_vector)
15:    < π*, Q > ← PIA(S, A, P, R, s0)     ▷ PIA returns offloading decision policy
16:    return < π*, Q >
17: end procedure
```

Algorithm 2 Edge Offloading Process

```
1: procedure EDGE_OFF_PROC(train_dataset, tasks)
2:     S ← (q_md, q_ed, q_ec, q_ed, q_cd)                    ▷ Offloading sites
3:     A ← (a_md, a_ed, a_ec, a_ed, a_cd)                    ▷ Action decisions
4:     < π*, Q > ← EDGE_OFF_ALGO(S, A, train_dataset, tasks)
5:     for each state s in S do
6:         a ← π*(s)                                  ▷ for state s get best action a
7:         while True do
8:             if λ_T(s,a) then    ▷ if offloading fails then consider another action a
9:                 Q ← Q − {(s, a)}
10:                if Q = ∅ then return "No feasible solution"
11:                end if
12:                a ← argmax_a[Q(s, a)]                  ▷ get next best action a
13:                continue
14:            else
15:                ω = ω + {(s, a)}                       ▷ store feasible action a
16:                break
17:            end if
18:        end while
19:    end for
20:    return ω                                  ▷ return feasible offloading policy
21: end procedure
```

3.1.3 Failure Monitor. Failure monitor collects historical system trace logs on remote offloading sites for availability estimation. We employ heartbeat failure detection [1] to collect traces. This approach sends ping messages to remote offloading sites at a fixed time interval. Offloading site is considered to be unavailable if the ping is not answered before timeout. Recommended configuration settings for heartbeat protocols are time intervals of 150 ms and 10 timeouts [1]. Therefore, the offloading site is considered to be unavailable after 1.5 seconds, which captures the network variability due to different network delays between nodes.

3.1.4 Service Availability Estimator. We select the SVR algorithm for availability predictions, which provides prediction accuracy above 90% [15] and requires a small training dataset [6] as opposed to deep neural networks. The algorithm takes as input historical failure traces as input and its accuracy depends on hyper-parameters C and ϵ. Due to the near real-time requirements of our scenario, we use [5] parameter selection algorithm to reduce response time. C is defined in Equation 1 and ϵ in Equation 2,

$$C = max(|\bar{y} + 3\sigma|, |\bar{y} - 3\sigma|) \tag{1}$$

$$\epsilon = 3\sigma\sqrt{\frac{ln(m)}{m}} \tag{2}$$

where y is availability dataset, $\bar{y}$ represents the arithmetic mean, m is a dataset sample size and σ represents the standard deviation of the dataset. As a kernel solution, we use the Gaussian RBF kernel function which can estimate time-series data that exhibit non-linear behavior such as failures.

3.2 Proposed Method

3.2.1 MDP offloading model. We employ offloading MDP in [31], which is defined as a labeled transition system with: (i) state-space $S = \{MD, ED, EC, ER, CD\}$ representing offloading site where a current task is offloaded, (ii) action set $A = \{MD, ED, EC, ER, CD\}$ represents the offloading site where to offload next task, (iii) P probabilistic state transition matrix representing offloading service availability, and (iv) R matrix of rewards associated with RT and BL. The goal is to maximize rewards by minimizing RT and maximizing BL. We use Policy Iteration Algorithm (PIA) [17] to iterate MDP and find a feasible offloading policy.

3.2.2 Edge Offloading Algorithm. The Algorithm 1 describes the edge offloading while Algorithm 2 executes offloading decisions. In Algorithm 1, the loop on lines 4-9 iterates over application tasks and computes BL and RT for each offloading site. In line 12, the SVR algorithm estimates offloading service sites' availability, based on the probability matrix P which is constructed on line 13. On line 14, the reward matrix R is computed and forwarded together with MDP's states, actions, and P to PIA, which synthesizes the offloading policy π (line 15). Policy π is executed during runtime by the Algorithm 2. Within the for loop (lines 5-19) offloading is performed. If the target offloading site fails during runtime, offloading is classified as failed (line 8) and the next alternative is considered (line 12). The algorithm terminates when offloading is successful (lines 15-16) and returns a feasible offloading policy (line 20) or when no service site is available (line 10) and returns an error message.

4 PROTOTYPE IMPLEMENTATION

4.1 Cluster Networking

The Raspberry Pi (RPi) single-hop away edge nodes provide wireless connectivity to nearby mobile devices. Configuration requires installation of local DHCP and DNS servers which provide control over mobile IP address space.

Deploying the Kubernetes cluster over the public and private IP subnets is not straightforward. To address firewall

Table 1: Experimental Setup

Hardware specifications			
Node Type	**CPU**	**RAM [GB]**	**STORAGE [GB]**
Huawei P Z (mob.)	Quad-core ARM Cortex-A53 1.7 GHz	4	64
RPi 3B+ (master)	Quad-core ARMv7 at 1.4GHz	1	64
RPi 3B+ (ED)	Quad-core ARMv7 at 1.4GHz	1	64
RPi 3B+ (EC)	Quad-core ARMv7 at 1.4GHz	1	64
RPi 3B+ (ER)	Quad-core ARMv7 at 1.4GHz	1	64
AMD64 (cloud)	48-core Intel Xeon E5-2650 v4 @ 2.2GHz	128	1000

and NAT translation issues, we deploy the private virtual networking solution called OpenVPN, which provides point-to-point communication and shared virtual IP address space.

4.2 Micro-service Containerization

We developed our microservices using Python 3.6 programming language and containerized them using Docker. We use the buildx command-line interface (CLI) plugin that utilizes machine processor emulator QEMU to build a common Docker container image for both CPU architectures available in the cluster, i.e., RPi ARMv7 and AMD64.

Micro-services on the mobile device are developed using Python Kivy mobile cross-platform framework We developed it as a Python application for Android OS mobile devices. These microservices do not have to be containerized. However, microservices can be placed on the dedicated offloading site instead (as part of the Kubernetes cluster) to reduce mobile devices' resource consumption.

4.3 Service Deployment

Since offloading requests are performed by mobile devices through HTTP, we deploy Flask web service on each offloading site. Flask provides necessary web services without additional third-party components. We instantiate it as an additional microservice on the remote offloading site, together with the failure monitor and prediction engine, on a single Kubernetes pod. Each pod has its unique virtual IP address dispatched by the Flannel Container Networking Interface (CNI) plugin. We also employ the NGINX reverse proxy to redirect HTTP requests to appropriate offloading services. Combining NGINX web service on the Kubernetes cluster level with Flask micro web services on the offloading site, we can expose offloading sites to mobile devices.

5 EXPERIMENTAL EVALUATION

5.1 Experimental Setup

We evaluate our edge offloading framework on the test-bed described in Table 1. Infrastructure setup is summarized in Figure 2: Huawei P Smart Z is a mobile device; RPis are edge nodes, deployed as in Figure 3b and AMD64 in Figure 3a is used to simulate a cloud data center. Resource heterogeneity

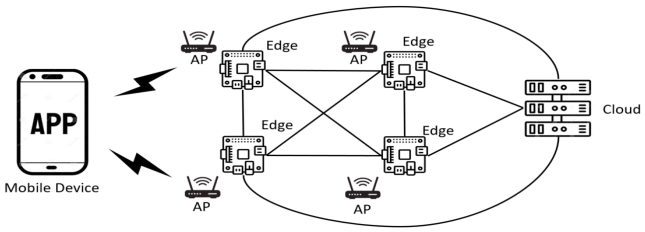
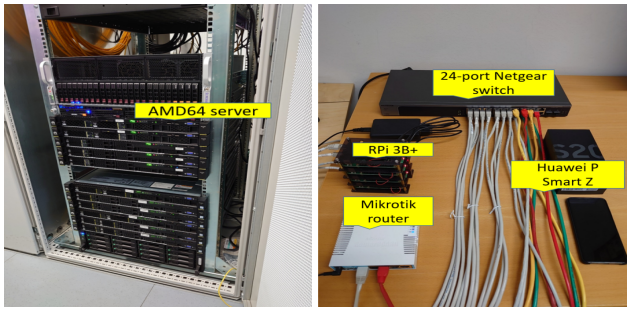


Figure 2: Infrastructure Overview



(a) AMD64 Cloud Server **(b) Edge Infrastructure**

Figure 3: Hardware Infrastructure for the Experiments

is simulated by defining hardware and network limitations, as in [31]. They are parameterized in the clusters' PostgreSQL database as experimental input parameters. When offloading micro-service is deployed on the Kubernetes cluster, it connects to the database and retrieves the resource information based on which resource capacity of the underlying site is specified.

Edge nodes and the cloud server are integrated into a single Kubernetes cluster while a mobile device is implemented as an external user. One of the RPi edge nodes is configured as a master node and the other nodes are configured as worker nodes where offloading micro-services are deployed and implemented as Docker containers. They are deployed according to the node labeling system. Each node in the Kubernetes cluster has a certain label that represents a node type. For instance, if we want to mark a certain node as an edge database server for handling data-intensive applications, the node is labeled as edge database, and inserted into Kubernetes deployment manifest file.

The mobile applications used in the evaluation are Directed Acyclic Graphs (DAGs) taken from [7, 31], namely (i) *Facebook*, (ii) *GPS navigation*, (iii) *Facerecognizer*, (iv) *Antivirus*, and (v) *Chess*. The mobile applications are sampled according to a probability distribution taken from [8]. The simulated workload is utilized since the real application would require application partitioning and profiling mechanisms,

which are out of the scope of this paper. To offload the simulated DAG workload on the remote Kubernetes offloading service site, the JSON serialization is performed. It converts task objects into byte strings which are necessary to transfer the data via a network to the target offloading service site. On the recipient site, JSON deserialization is performed to acquire the original task object from which it extracts all necessary resource information.

To simulate failures on remote offloading sites, we implement a two-state Markov state machine. This kind of on/off (failure/non-failure) model is used to simulate network intermittent channels where simplicity is preferred over complexity [4]. The probabilistic availability distribution is extracted from the local failure dataset Los Alamos National Laboratory (LANL) for HPC clusters [22]. We adopted this dataset since it shares some characteristics with edge computing, i.e., distributed architecture, a large number of nodes, and heterogeneous resources. Possible limitation of using the HPC dataset for the edge is that it probably cannot replicate the edge behavior completely. HPC cluster nodes usually have superior resources, equipped with additional support systems (e.g. fan units, backup power generators) and interlinked with high-speed network connections where in edge could not be the case. We pick several nodes from the dataset to compute availability distributions for each offloading service (Table 2). The nodes are categorized according to their availability levels as low (LA), medium (MA), and high (HA) based on failure rates, and mean and deviation of their availability distribution. Their hardware characteristics are the second selection criteria. For instance, nodes from systems 5 and 7 are selected for the ED edge node due to a large number of nodes (larger data storage). The nodes are named <*systemID_nodenumber*> where both index numbers are obtained from the original dataset. They are split into train and test data in a proportion of 80%-20% as the general rule of thumb practiced in ML community. The nodes from systems 5 and 7 are most suitable to the ED edge node due to a large number of nodes (larger data storage). The EC node is sampled from nodes of systems 19 and 20 which have a higher ratio of processors per node (higher computational power). ER edge node is sampled from 3, 4, and 16 systems due to a lower processor per node ratio, a minimum quantity of network interface cards, and a moderate number of nodes compared relatively to the ED and the EC nodes. The cloud is sampled only from 22 system since it has a single node with the highest processor per node ratio and RAM capacity in the entire dataset.

For statistical significance, we set application runs to 1000 and average results of 100 executions. Results are compared with the solution in [31], which emulates default Kubernetes

Table 2: Dataset configurations

	Dataset configurations				
Service	**DS1**	**DS2**	**DS3**	**DS4**	**DS5**
ED	HA (7_1)	MA (5_158)	HA (5_165)	HA (5_243)	HA (5_48)
EC	LA (19_1)	MA (19_11)	MA (19_4)	HA (19_8)	HA (20_41)
ER	HA (3_0)	HA (16_80)	MA (4_55)	MA (4_1)	HA (4_3)
CD	HA (22_0)	HA (22_0)	HA (22_0)	HA (22_0)	HA (22_0)

greedy multi-criteria decision-making (with adjusted parameter tuning) and estimates availability levels through mean-time-between-failures (MTBF). Moreover, it is augmented with re-computing and check-pointing and named KubeHybrid as a Kubernetes hybrid (proactive-reactive) decision-maker. The source is available online[1].

5.2 Results

Figures 4, 5 and 6 illustrates results for RT, BL and availability respectively. Our solution outperforms the KubeHybrid in all three objectives. There is a strong correlation between the three objectives since higher service availability increases BL and decreases RT. This is explained by the necessity of re-transmitting offloading tasks in case of offloading failures, which consumes additional mobile devices' resources. Hence, higher availability ensures more BL and shorter RT. In our evaluation, we consider also offloading distribution, i.e., the number of tasks offloaded per offloading service site.

Figures 4, 5 and 6 depict that for DS1 configuration our solution achieves around 600 seconds RT, 98.4% BL, and 99.6% availability against the KubeHybrid with 760 seconds RT, 98.15% BL, 98.6% availability. According to offloading distribution, our solution offloaded around 50% of tasks to EC, completely avoiding Cloud (0% distribution) while ER receives less than 0.1% distribution. Other tasks are offloaded either on a mobile device or an ED service. Despite lower availability, the prediction engine predicts service availability accurately enough to select EC service for timely task offloading. Moreover, 50% implies that not only CI-intensive tasks are offloaded but also moderate tasks. This is because ER has a lower CPU than EC. The KubeHybrid algorithm, on the other hand, relies on a cloud distribution of 2.9%, while edge services are consumed proportionally to their resource availability. ED is the most used (31.7%), ER is moderately utilized (17.8%) while EC is the least used edge service (7.9%). KubeHybrid depends on an average MTBF, which reduces the prediction accuracy. The SVR algorithm, on the other hand, generally, did yield in our experiment the prediction accuracy between 55% and 90% measured in R2, so-called the good-of-fitness metric. It is widely used in statistics to

[1]https://github.com/jzilic1991/edge-offloading/tree/master

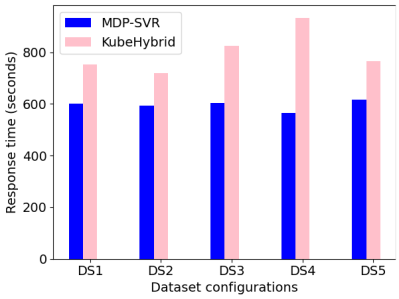


Figure 4: Application response time

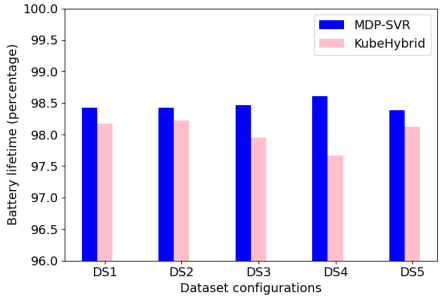


Figure 5: Mobile battery lifetime

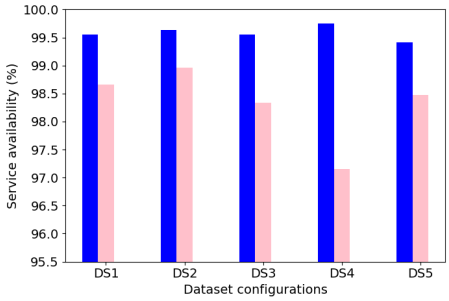


Figure 6: Service availability

measure accuracy in predicting future outcomes and usually preferred since it is more intuitive and informative then other metric alternatives.

For DS2 and DS5 configurations, our solution achieves offloading distribution and prediction accuracy similar to DS1, which indicates adaptability towards different availability distributions. However, in DS4 configuration ED is the most utilized service, with 37% offloading distribution, due to its high availability and resource capabilities. The second most utilized service is EC since it has more hardware capabilities than ER service. In our approach, none of the tasks are offloaded to the Cloud. The KubeHybrid approach, instead, prefers ED service the most (26%) but cloud service is the second most utilized (15%). When ED service is unavailable, data intensive tasks are offloaded to the cloud. However, the higher latency results in its worst performance of around 950 seconds RT, 97.5% BL, and 97% availability.

6 RELATED WORK

Mostly reactive failure management techniques has been discussed in the related edge computing literature thus far. The authors in [12] perform container checkpointing at the edge to ensure high service availability while [16] checkpoints the applications offloaded on the offloading sites. Another work [27] locally re-computes offloaded tasks on a mobile device when task offloading fails. Research conducted both in simulated [7, 8, 11] and real-world edge environment [26] do not consider proactive failure mitigation. Failure prediction approaches such as [6, 15] proved the effectiveness of proactive failure management, but these approaches are neither applied at the edge nor on a real-world test-bed.

There exists few studies focusing on proactive failure management. They propose risk based [23], learning based [2, 3], or formal verification based [31] solutions. Nevertheless, none of these consider microservices. We summarise our literature review in Table 3. The works are selected according to whether they focus on microservice architecture (MSA), edge offloading (OFF), proactive failure prediction (PRO), container orchestration (ORCH), and real-world implementation (REAL). We conclude that to the best of our knowledge, none of the selected works covers all aforementioned objectives.

Table 3: Overview of state-of-the-art literature

Publication	MSA	OFF	PRO	ORCH	REAL
Suk et al. [23]			✓	✓	✓
Aral et al. [2]			✓		
Zilic et al. [31]		✓	✓		
Dupont et al. [9]		✓		✓	✓
Tang et al. [26]	✓	✓		✓	
Wu et al. [29]	✓			✓	✓
Samanta et al. [20]	✓			✓	
This work	✓	✓	✓	✓	✓

7 CONCLUSION AND FUTURE WORK

We designed a proactive fault-tolerant edge offloading microservice which allows to reduce application response time and increase mobile battery lifetime. Our solution outperforms default Kubernetes scheduler, augmented with hybrid fault tolerance. Experimentation was conducted on a real-world edge-cloud testbed and showed great promise for the failure prediction in edge offloading. The web link to the experiments' source code is provided in the paper.

In the future, we plan to apply runtime failure injection to evaluate edge offloading performance under stress instead of the two state model used in this work. Utilisation of edge-related traces for the evaluation of the approach would strengthen the evaluation. Operating computation-intensive software, such as a decision engine, on the mobile device can hinder offloading benefits. As a consequence, we will investigate placing the decision engine at the Edge. Infrastructure providers might deploy more powerful edge nodes (i.e., micro data centers) to address lower reliability of RPis.

ACKNOWLEDGEMENTS

This work is partially funded by Rucon project (Runtime Control in Multi Clouds), FWF Y 904 START-Programm 2015.

REFERENCES

[1] [n.d.]. Network Heartbeat Configuration. https://www.aerospike.com/docs/operations/configure/network/heartbeat/. Accessed: 2020-09-02.

[2] Atakan Aral and Ivona Brandić. 2018. Dependency mining for service resilience at the edge. In *2018 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE, 228–242.

[3] Atakan Aral and Ivona Brandić. 2020. Learning spatiotemporal failure dependencies for resilient edge computing services. *IEEE Transactions on Parallel and Distributed Systems* 32, 7 (2020), 1578–1590.

[4] Fulvio Babich and Giancarlo Lombardi. 2000. A Markov model for the mobile propagation channel. *IEEE Transactions on Vehicular Technology* 49, 1 (2000), 63–73.

[5] Vladimir Cherkassky and Yunqian Ma. 2004. Practical selection of SVM parameters and noise estimation for SVM regression. *Neural networks* 17, 1 (2004), 113–126.

[6] Márcio das Chagas Moura, Enrico Zio, Isis Didier Lins, and Enrique Droguett. 2011. Failure and reliability prediction by support vector machines regression of time series data. *Reliability Engineering & System Safety* 96, 11 (2011), 1527–1534.

[7] Vincenzo De Maio and Ivona Brandic. 2018. First hop mobile offloading of dag computations. In *2018 18th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGRID)*. 83–92.

[8] Vincenzo De Maio and Ivona Brandic. 2019. Multi-objective mobile edge provisioning in small cell clouds. In *2019 ACM/SPEC International Conference on Performance Engineering*. 127–138.

[9] Corentin Dupont, Raffaele Giaffreda, and Luca Capra. 2017. Edge computing in IoT context: Horizontal and vertical Linux container migration. In *2017 Global Internet of Things Summit (GIoTS)*. IEEE, 1–4.

[10] Che-Wei Hsu, Yung-Lin Hsu, and Hung-Yu Wei. 2020. Energy-efficient edge offloading in heterogeneous industrial IoT networks for factory of future. *IEEE Access* 8 (2020), 183035–183050.

[11] Miao Hu, Zixuan Xie, Di Wu, Yipeng Zhou, Xu Chen, and Liang Xiao. 2020. Heterogeneous edge offloading with incomplete information: A minority game approach. *IEEE Transactions on Parallel and Distributed Systems* 31, 9 (2020), 2139–2154.

[12] Bukhary Ikhwan Ismail, Ehsan Mostajeran Goortani, Mohd Bazli Ab Karim, Wong Ming Tat, Sharipah Setapa, Jing Yuan Luke, and Ong Hong Hoe. 2015. Evaluation of docker as edge computing platform. In *2015 IEEE Conference on Open Systems (ICOS)*. IEEE, 130–135.

[13] Ivan Lujic, Vincenzo De Maio, Srikumar Venugopal, and Ivona Brandic. 2021. SEA-LEAP: Self-adaptive and Locality-aware Edge Analytics Placement. *IEEE Transactions on Services Computing* (2021), 12 pages.

[14] Pavel Mach and Zdenek Becvar. 2017. Mobile edge computing: A survey on architecture and computation offloading. *IEEE Communications Surveys & Tutorials* 19, 3 (2017), 1628–1656.

[15] Bashir Mohammed, Irfan Awan, Hassan Ugail, and Muhammad Younas. 2019. Failure prediction using machine learning in a virtualised HPC system and application. *Cluster Computing* 22, 2 (2019), 471–485.

[16] Shumao Ou, Yumin Wu, Kun Yang, and Bosheng Zhou. 2008. Performance analysis of fault-tolerant offloading systems for pervasive services in mobile wireless environments. In *2008 IEEE International Conference on Communications*. 1856–1860.

[17] Martin L Puterman. 2014. *Markov decision processes: discrete stochastic dynamic programming*. John Wiley & Sons.

[18] Guanjin Qu, Huaming Wu, Ruidong Li, and Pengfei Jiao. 2021. Dmro: A deep meta reinforcement learning-based task offloading framework for edge-cloud computing. *IEEE Transactions on Network and Service Management* 18, 3 (2021), 3448–3459.

[19] Aki Saarinen, Matti Siekkinen, Yu Xiao, Jukka K Nurminen, Matti Kemppainen, and Pan Hui. 2012. Can offloading save energy for popular apps?. In *Seventh ACM international workshop on Mobility in the evolving internet architecture*. 3–10.

[20] Amit Samanta and Jianhua Tang. 2020. Dyme: Dynamic microservice scheduling in edge computing enabled IoT. *IEEE Internet of Things Journal* 7, 7 (2020), 6164–6174.

[21] Mahadev Satyanarayanan, Paramvir Bahl, Ramón Caceres, and Nigel Davies. 2009. The case for vm-based cloudlets in mobile computing. *IEEE pervasive Computing* 8, 4 (2009), 14–23.

[22] Bianca Schroeder and Garth A Gibson. 2009. A large-scale study of failures in high-performance computing systems. *IEEE transactions on Dependable and Secure Computing* 7, 4 (2009), 337–350.

[23] Tonghoon Suk, Jinho Hwang, Muhammed Fatih Bulut, and Zemei Zeng. 2019. Failure-aware application placement modeling and optimization in high turnover DevOps environment. In *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. 115–123.

[24] Noriyuki Takahashi, Hiroyuki Tanaka, and Ryutaro Kawamura. 2015. Analysis of process assignment in multi-tier mobile cloud computing and application to edge accelerated web browsing. In *2015 3rd IEEE International Conference on Mobile Cloud Computing, Services, and Engineering*. IEEE, 233–234.

[25] Yongmin Tan, Hiep Nguyen, Zhiming Shen, Xiaohui Gu, Chitra Venkatramani, and Deepak Rajan. 2012. Prepare: Predictive performance anomaly prevention for virtualized cloud systems. In *2012 IEEE 32nd International Conference on Distributed Computing Systems*. 285–294.

[26] Jie Tang, Rao Yu, Shaoshan Liu, and Jean-Luc Gaudiot. 2020. A container based edge offloading framework for autonomous driving. *IEEE Access* 8 (2020), 33713–33726.

[27] Qiushi Wang, Huaming Wu, and Katinka Wolter. 2013. Model-based performance analysis of local re-execution scheme in offloading system. In *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. 1–6.

[28] Kwame-Lante Wright, Ashiwan Sivakumar, Peter Steenkiste, Bo Yu, and Fan Bai. 2020. Cloudslam: Edge offloading of stateful vehicular applications. In *2020 IEEE/ACM Symposium on Edge Computing (SEC)*. IEEE, 139–151.

[29] Li Wu, Johan Tordsson, Alexander Acker, and Odej Kao. 2020. MicroRAS: Automatic Recovery in the Absence of Historical Failure Data for Microservice Systems. In *2020 IEEE/ACM 13th International Conference on Utility and Cloud Computing (UCC)*. 227–236.

[30] Changsheng You, Kaibin Huang, Hyukjin Chae, and Byoung-Hoon Kim. 2016. Energy-efficient resource allocation for mobile-edge computation offloading. *IEEE Transactions on Wireless Communications* 16, 3 (2016), 1397–1411.

[31] Josip Zilic, Atakan Aral, and Ivona Brandic. 2019. EFPO: Energy efficient and failure predictive edge offloading. In *12th IEEE/ACM International Conference on Utility and Cloud Computing*. 165–175.