Probabilistically Reliable and Efficient Edge Offloading

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

Atakan Aral
Vienna University of Technology
Vienna, Austria
atakan.aral@tuwien.ac.at

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

ABSTRACT

Edge offloading as a concept offloads mobile applications on remote Edge and Cloud servers to reduce application response time and battery consumption on the mobile device. However, Edge Computing as infrastructure is still not standardized, which implies that Edge devices are diverse in resource and availability characteristics. This impacts the Edge offloading control which has to consider a wide range of offloading sites to offload parts of the mobile application. Moreover, during offloading, failures that exhibit stochastic behavior can disrupt the process and result in unexpected costs. Most of the literature is focused on system performance optimization without considering the offloading failure impact on the system. The offloading performance optimization should be combined with the system reliability objective by predicting failures based on the historical data and manage offloading decision-making accordingly. Thus, we employ the Edge offloading framework that features both Markov Decision Process (MDP), that accounts for both the nondeterminism and the stochastic behavior of the Edge offloading, and the Support Vector Regression (SVR) algorithm with its prediction capabilities that are proved in the reliability engineering literature to yield promising results. The solution is verified via a simulation environment by using a real-world failure dataset from Los Alamos National Laboratory (LANL) and various types of mobile applications as Directed Acyclic Graphs (DAG). DAGs are sampled based on LiveLab application usage traces. Evaluation results show that our proposed solution can improve response time up to 48%, energy efficiency up to 41%, and reduce failure rates by 96%. compared to state-of-the-art offloading decision engines. We also provide a learning complexity analysis of the aforementioned algorithms to give an insight into algorithm execution feasibility under Edge Computing settings.

CCS CONCEPTS

• **Computer systems organization** → **Distributed architectures**; • **Computing methodologies** → *Modeling and simulation.*

KEYWORDS

edge offloading; learning complexity analysis; markov decision process; support vector regression;

Conference'17, July 2017, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

ACM Reference Format:
Josip Zilic, Atakan Aral, and Ivona Brandic. 2021. Probabilistically Reliable and Efficient Edge Offloading. In *Proceedings of ACM Conference (Conference'17).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The resource consumption of mobile applications is increasing on a larger scale and more rapidly than the hardware capabilities of mobile devices. This compels researchers and developers to find an alternative solution that is both technically and economically sustainable. Partitioning mobile applications into smaller chunks as application tasks and offloading them on the remote infrastructure proved to be a viable solution [22]. Cloud data center as a remote counterpart with superior resources appears as a prominent solution to execute offloaded application tasks. However, in some latency-sensitive applications such as live navigation, AR/VR, and smart grids, satisfying imposed requirements are not guaranteed due to large geographical distance and dense backhaul network traffic between end-users and Cloud data centers. Those strict requirements can be fulfilled by deploying Edge nodes in the proximity of the end-users instead of utilizing the remote Cloud nodes. This reduces geographical distance and alleviates network traffic. However, without considering the heterogeneous resource and reliability characteristics of the Edge Computing platform, failures can occur. They may prevent or postpone the offloading process leading to increased mobile device energy consumption and application response time [38, 39].

To manage the Edge offloading process, a variety of offloading sites with different resource characteristics, as well as different types of mobile applications have to be accounted for. Additionally, offloading sites exhibit different availability characteristics that can disrupt the offloading process in form of failures and produce unexpected system costs. It is preferable to predict failures rather than triggering reactive countermeasures after failures already happen. Thus, our solution consists of Markov Decision Process (MDP) that handles both the nondeterminism and the stochastic behavior of the Edge offloading process, and the Support Vector Regression (SVR) capable of predicting time-series reliability data. The SVR predictability performance of the time-series data has been proven to yield better results compared to other ML solutions [12]. Both methods are a joint solution proposal to handle the Edge offloading process with the objectives of reducing mobile application response time and energy consumption of the mobile device.

Offloading frameworks that are realized as MDP approaches are used in Mobile Cloud [31] and Mobile Edge environments [4] without considering the impact of offloading failures. Other MDP offloading frameworks that are considering offloading failures [38, 39]

are considering only link failures or using a general simple reliability model. The use of Machine Learning (ML), on the other hand, can increase prediction accuracy by exploiting the underlying datasets and capturing failures that exhibit stochastic and non-linear behavior [10, 12]. Therefore, we adopt MDP and SVR approaches to deliver efficient and reliable Edge offloading. We perform evaluation based on a real-world failure dataset of Los Alamos National Laboratory (LANL) and mobile applications are sampled from probability distributions computed from LiveLab application usage traces. Evaluation results show that our solution proposal can improve time performance up to 48%, energy efficiency up to 41%, and failure rates up to 96% compared to baseline offloading decision engines from the literature. Moreover, we will also provide learning complexity analysis of both mentioned algorithms to estimate computational overhead by using the Probably Approximately Correct (PAC) approach and the Vapnik–Chervonenkis (VC) dimension.

The paper is organized as follows. The background about used methodologies is described in Section 2. Section 3 provides a system and mathematical model of Edge offloading. Section 4 presents the evaluation results. In Section 5, we provide learning complexity analysis of used algorithms. Related work is discussed in Section 6. Finally, Section 7 mentions future work and concludes the paper.

2 BACKGROUND

2.1 Mobile Applications and Edge Offloading

Mobile applications are usually composed of smaller tasks that are mutually interdependent. Each task has its resource requirements including the offloadability indicator which indicates the ability of the task to be offloaded or not. Application tasks can be non-offloadable due to dependencies on some local physical functions (e.g. camera). Tasks can correspond to functions, classes, or threads. Due to its partitioned nature, a mobile application can be modeled as a Directed Acyclic Graph (DAG) [13]. The DAG model is composed of vertices and edges, where vertices represent the application tasks and edges represent the task inter-dependencies. DAG models are appropriate when the task execution order is relevant. A DAG example of an Antivirus application is illustrated in Figure 1. The DAG model consists of five tasks, where blue circles represent offloadable and red circles represent non-offloadable tasks.

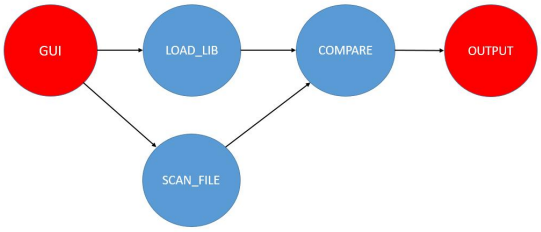


Figure 1: Antivirus mobile application

Edge offloading is a process where application tasks are offloaded from a mobile device to a remote infrastructure. The offloading is performed by a software unit called the offloading decision engine (ODE) which is usually executed on the mobile device. There is also the possibility that ODE is executed on a remote site to alleviate consumption on the mobile device. This option is discussed in the Subsection 5.3. ODE performs a decision-making function with assistance from the prediction engine (PE) which is positioned on each offloading site and predicts the future availability of the sites. Once offloading is completed, the infrastructure executes tasks and sends the output to its destination. This process repeats until the application terminates.

The Edge offloading model is illustrated in Figure 2. The main components of the PE engine are: (1) *Failure monitoring* which collects all failure traces logs from the sites' local storage (Step 1a) and prepares the data (Step 2a), (2) *SVR* outputs availability predictions based on training samples and forwards them it to the ODE decision engine for offloading optimization (Step 3). During the data pre-processing stage (Step 2a), failure trace logs are parsed and transformed into the availability distribution based on failure frequency occurrence and life span. The data samples are interpreted as the probability that the offloading site will be available for offloading and task execution. ODE architecture is similar to [39], which collects mobile application DAG structure and its resource requirements, and remote infrastructure resource capacities (Steps 1b, 1c, 2b, and 2c). Additionally, the offloading site availability predictions from the PE engine are forwarded to the ODE engine to output the offloading decision policy (Step 4) based on which offloading decisions are performed (Step 5).

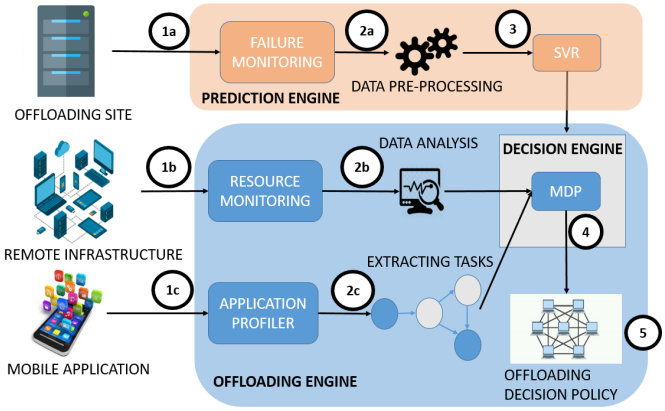


Figure 2: Edge offloading model

2.2 Support Vector Regression

SVR is a supervised ML algorithm that uses the regression model upon underlying data to predict future events and outcomes. It is based on the structural risk minimization (SRM) principle. It improves the ML generalization performance based on the trade-off between model accuracy during the training phase and can predict unseen values during the testing phase [34]. The SRM minimizes the upper bound on the generalization error and avoids overfitting. Therefore, it can outperform other ML algorithms in predicting the time-series data [12]. The main purpose of the SVR algorithm in this work is to output a regression model that fits the failure trace data to estimate the future offloading site availability. This information is

used as an input to the ODE engine for more informative offloading decision-making (Figure 2). The algorithm is instantiated on each offloading site and executed separately and independently. It parses failure trace logs as training samples. In case of failure on the offloading site, SVR is restarted and rebooted together with the failure trace logs from the local storage.

2.3 Markov Decision Process

MDP is a discrete stochastic optimization algorithm widely used in decision-making situations where the future outcomes are partly probabilistic and partly under the control of a decision-maker. It is defined as labeled transition system that consists of *states S*, *actions A* and *transitions T*. The state represents system configuration whereas transitions and actions define the behavior of the system. The model can be verified with certain numerical model-checking algorithms such as Value Iteration Algorithm (VIA) and Policy Iteration Algorithm (PIA) that outputs a *decision policy* in terms of best actions in the certain states. The MDP model has to be augmented with *reward functions R* to compute the aforementioned policy. The general advantage of this approach is automatic and exhaustive state-space search. However, in the case of larger systems, this may result in a state-space explosion phenomenon that exponentially prolongs the model-checking process. Theorem proving, as an alternative approach, can work with more accurate representations of the system but it is a more time-consuming process and manually obtained.

2.4 Learning Complexity Analysis

Learning complexity analysis provides the theoretical insight into the computational overhead of employed solutions in this work. It can express the computational and the sample complexity in terms of training sample sizes that the ML algorithm requires to learn a given problem. For this, we use the Probably Approximately Correct (PAC) learning. It is a mathematical framework that estimates the sample complexity of the ML algorithm. The goal is to determine the upper bound on a training sample size that the ML algorithm requires to learn or solve the problem. The PAC logic is to select a hypothesis function h with a high probability that will have a low generalization error by approximating the target concept function c which represents the true behavior of the system.

As a supplement to the PAC analysis, the VC dimension expresses the capacity of the underlying model as the cardinality of the maximum number of data points that can be shattered by the same model for any combination of the labels associated with those points. It is often annotated as a $VC(H)$ for a specific hypothesis space H. This is beneficial when the hypothesis space H goes to infinity due to infinite possible combinations of mapping input-output pairs. By applying the shattering concept, the VC dimension upper bounds the sample complexity. Alternative approaches to PAC and VC are total mistake bound model [20] and weighted majority method [21] but they require parameter tuning and learning performance measures.

3 MDP-SVR OFFLOADING FRAMEWORK

3.1 System Architecture and MDP model

We adopt the architecture from [39]. This fits our needs since it can capture (i) the Edge resource heterogeneity and the availability diversity, (ii) interplay between a mobile device, an Edge layer, and a Cloud data center, and (iii) the mesh network topology to support the system robustness against failures. In the proposed architecture, the Edge resources are diversified between three Edge server types: (i) *Edge database server* (ED) which has large data storage capabilities and fast network transmission rates for handling data-intensive (DI) applications such as Antivirus scanning software or image processing application, (ii) *Edge computational server* (EC) has greater computational power that is suitable for computational-intensive (CI) applications such as strategic games with the AI support, and (iii) *Edge regular server* (ER) has intermediate resources suitable for applications that do not require a large amount of computation or data storage capacities, such as GPS navigation or posting on Facebook.

To fit the Edge offloading model from Section 2.1, we created an MDP model where offloading sites are modeled as states S, offloading decisions as actions A, offloading site availability as probabilistic transitions T, model iterations as discrete-time epochs t when the offloading decision is triggered, and the mobile device energy consumption and the application response time as reward functions R. Verifying the MDP model by numerical model-checking solution yields a near-optimal offloading decision policy $\pi*$ based on which MDP decision-maker gives an offloading decision. Works like [4, 31, 39] used the Value Iteration Algorithm (VIA) as a numerical model-checking algorithm due to theoretical simplicity and ease of implementation. We use the Policy Iteration Algorithm (PIA) since policies can converge in much fewer iterations than state values as in the VIA algorithm.

3.2 SVR Modelling

The main task of the SVR algorithm is to predict the unknown real-valued function based on the past values. The prediction procedure selects the most appropriate hypothesis function h from a set of approximating functions H. It is selected by the quality of prediction measured by the loss function $L(y, \hat{y})$, where y represents the actual value and $\hat{y}$ represents the predicted value. The loss function commonly used for the SVR regression is the ϵ-insensitive loss function proposed by [32]:

$$L(y, \hat{y}) = \begin{cases} 0 & if |y - \hat{y}| \leq \epsilon \\ |y - \hat{y}| - \epsilon & otherwise \end{cases} \quad (1)$$

The SVR prediction performance strictly depends on hyperparameters known as C and ϵ. C is a regularization parameter that models the ability to generalize the unseen data as a trade-off between the training and testing phases. ϵ parameter determines the level of regressor accuracy by controlling the width of the ϵ-insensitive area in the loss function $L(y, \hat{y})$ as presented in the Equation 1. Both parameters in the SVR implementation in various software libraries are often determined as user-defined input parameters. Selecting both parameters carefully can boost or undermine SVR prediction performance. In the literature, there is a large body of work that is focusing on optimizing both parameters

to increase SVR performance by applying different optimization algorithms such as genetic [10] and distribution algorithms [18]. Instead of using a complex optimization algorithm which increases the computational overhead drastically, we use the simple parameter selection algorithm proposed by [11]. The benefits of this kind of approach are simplicity, near-optimal performance, usability in different application domains, and applicability for various target functions and sample sizes. Both parameters are derived directly from the data and their formal definitions are:

$$C = max(|\bar{y} + 3\sigma|, |\bar{y} - 3\sigma|) \tag{2}$$

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

where $\bar{y}$ represents the arithmetic mean and σ represents the standard deviation of the y data. In this work, the y data represents offloading site availability.

3.3 Offloading Site Availability Model

The offloading site availability definition is expressed as a ratio between the uptime and the total time:

$$A = \frac{uptime}{uptime + downtime} \tag{4}$$

The characteristics like distributed architecture, absence of support systems (power supplies, cooling equipment, etc.) and erratic workload can be significant factors that contribute to the failure-prone Edge infrastructure. By using the SVR model from the Subsection 3.2, we can predict the future offloading site availability and mitigate offloading failures before the task offloading is performed. The offloading site availability can be formally described as a A_t availability probabilistic distribution which is defined as follows. Equation (5) defines a operational state $X(t)$ as a function that indicates the offloading site availability as a downtime or uptime in the t time moment:

$$X(t) = \begin{cases} 1 & uptime \\ 0 & downtime \end{cases} \tag{5}$$

The availability $A(t)$ in Equation (6) is defined probabilistically as the expectation that the offloading site will be in the uptime operational state $X(t)$ at time t:

$$A(t) = Pr[X(t) = 1] = E[X(t)] \tag{6}$$

And consequently, Equation (7) integrates the offloading site availability A_t as a probability distribution over time as defined:

$$A_t = \int_0^t A(t)dt \tag{7}$$

The SVR model computes the availability estimation of the offloading site and forwards it to the ODE unit on the mobile device to compute the offloading decision policy. An example of SVR predicting offloading site availability is illustrated in Figure 3. x and y axis represent time units (days) and availability values $A(t)$ respectively, whereas the pink curve is the real data and the green curve is the predicted data generated by the SVR algorithm. The example is taken from a LANL dataset for a single node. This is a representative example of availability distribution where nodes or services tend to be available 100% to maximize utilities and economic gains. However, due to severe failures, the availability can drop significantly and contain unavailability spikes that can last from hours to days. This supports the assumption that failures exhibit the non-linear behavior and hence the Gaussian RBF kernel function is a more justifiable kernel solution for the SVR algorithm than a linear or a polynomial kernel.

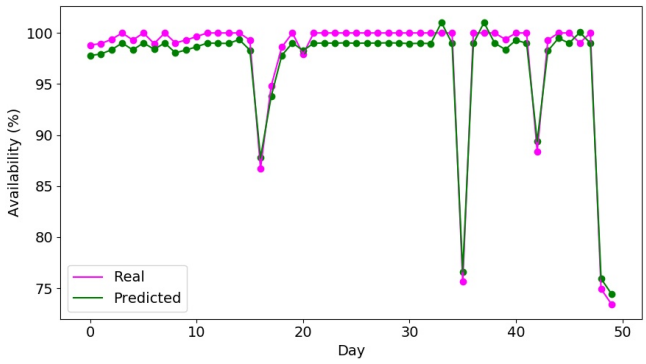


Figure 3: SVR offloading site availability prediction

3.4 Edge Offloading Model

The offloading site is defined as the resource capacity vector $q = (f, ram, storage, b, l)$, where f is the CPU computation power in millions instructions per second MIPS, ram is the memory capacity in GB, $storage$ is the data storage capacity in GB, b is the network bandwidth in Mbps and l is the network latency in ms. The application task is defined as the resource requirement vector $v = (\omega, ram, d_{in}, d_{out}, off)$, where ω is the CPU computation power in millions of instructions MI, ram is the memory consumption in GB, d_{in} is the input data size in KB, d_{out} is the output data size in KB and off indicates binary task offloadability.

The task offloading of the same task v can be repeated more than once during single mobile application execution under the condition that failure was observed during the offloading process. The process is repeated until the alternative offloading solution is found that mitigated the failure. To classify the offloading attempt as valid, the resource constraints of the offloading sites has to be respected. The following conditions has to be fulfilled:

(1) $\sum_{v \in V_q(t)} (ram_v) \leq ram_q$
(2) $\sum_{v \in V_q(t)} (d^v_{in} + d^v_{out}) \leq storage_q$
(3) $\delta_{in}(v) = \emptyset$
(4) $off_v = 1$

Cases (1) and (2) validates that all application tasks offloaded on the offloading site q in the discrete-time epoch t, denoted as $V_q(t)$, do not exceed RAM and data storage capacities. Moreover, the application task v to be offloaded, must not have any remaining task input dependencies $\delta_{in}(v)$ to other preceded tasks before offloading, according to case (3), and must be annotated as offloadable task off_v as in case (4).

3.5 Response Time Model

The response time model consists of the local computation time t_c, the uploading data transfer time t_u and the downloading data transfer time t_d. The local computation time is formally defined as a ratio between the computational amount ω_v of the application task v and the CPU frequency f_q of the offloading site q:

$$t_c(v,q) = \frac{\omega_v}{f_q}, \forall v \in V, \forall q \in Q \tag{8}$$

where V is denoted as a set of tasks from particular mobile application, whereas Q represents a set of offloading sites. Upload and download communication time is defined as time consumed for data transfer between the source q_i and the destination q_j offloading sites:

$$t_u(v,q_i,q_j) = \frac{d_u^v}{b_{q_iq_j}} + l_{q_iq_j}, \forall v \in V, \forall q_i,q_j \in Q, i \neq j \tag{9}$$

$$t_d(v,q_i,q_j) = \frac{d_d^v}{b_{q_iq_j}} + l_{q_iq_j}, \forall v \in V, \forall q_i,q_j \in Q, i \neq j \tag{10}$$

where $b_{q_iq_j}$ and $l_{q_iq_j}$ represents bandwidth and latency, and d_u^v and d_d^v represent upload and download data sizes respectively for the application task v. Based on Equations (8), (9) and (10), the task response time t_v is defined as:

$$t_v(v,q_i,q_j) = t_u(v,q_i,q_j) + t_c(v,q_j) + t_d(v,q_j,q_i) \tag{11}$$

If successive application tasks are offloaded and executed on the same offloading site, i.e., $q_i = q_j$, then the equation above is transformed into $t_v(v,q_i,q_j) = t_c(v,q_j)$ where uploading and downloading data transfers are omitted. When the entire mobile application is executed, then the application response time t_V is formally defined at top of Equation (11):

$$t_V = \sum_{v \in V} \sum_{q_i \in Q} \sum_{q_j \in Q} off(v,q_i,q_j) \times t_v(v,q_i,q_j) \tag{12}$$

where $off(v,q_i,q_j)$ represents the binary function that verifies whether application task v is offloaded from the source site q_i to the destination site q_j. Chaining different mobile applications into single execution sequence is defined in Equation 13. This is similar to work [14] where mobile workload model is introduced as a set of mobile applications prepared for execution. Thus, we are introducing W as a workload chain of multiple mobile applications and measuring the total response time t_W:

$$t_W = \sum_{V \in W} t_V \tag{13}$$

3.6 Energy Consumption Model

The energy consumption model is defined analogously to response time model Equation (11). The energy consumption is considered only from a mobile device perspective since energy supplies on the infrastructure are perceived as unlimited. The energy consumption model of the mobile device e_v is defined as:

$$e_v(v,q) = t_c(v,q) \times p_c + t_u(v,q) \times p_u + t_d(v,q) \times p_d \tag{14}$$

where analogously to Equations (12) and (13) we can measure over entire workload chain W as:

$$e_V = \sum_{v \in V} \sum_{q_i \in Q} \sum_{q_j \in Q} off(v,q_i,q_j) \times e_v(v,q_i,q_j) \tag{15}$$

$$e_W = \sum_{V \in W} e_V \tag{16}$$

where p_c is the mobile power consumption for the local computation, p_d is the mobile power consumption when downloading data and p_u is the mobile power consumption when uploading data. If two successive tasks are executed on the mobile device then equation above is transformed in $e_v(v,q) = t_c(v,q) \times p_c$ where power consumption for transferring data is omitted. In the case that mobile device only offloads task on the remote infrastructure, then the energy consumption accounts only for upload data transmission $e_v(v,q) = t_u(v,q) \times p_u$. Last case is when application task v is executed on the remote infrastructure without interaction with the mobile device. Although, the mobile device is not performing task execution, it still maintains the operational state and enters in the idle mode where power saving modes are activated to reduce the energy consumption. This is formally described as $e_v(v,q) = t_u(v,q) \times p_{idle}$ where p_{idle} is the power consumption in the idle mode. The assumption about the mobile power parameters when computing the energy consumption cost is considered as $p_u > p_d > p_c > p_{idle}$ based on previous work [19].

3.7 Failure Detection Model

The most widely used concept for the failure detection strategy is a heartbeat protocol. It sends ping messages every fixed time interval to other offloading sites. If the message is not received after the timeout period then the suspicious offloading site q is added to the list of suspected sites. When the message is received, then consequently, the site q is removed from the list. This approach is responsible for maintaining the integrity of the entire system architecture. The research about heartbeat improvements is reviewed in [7], but we use real-world implementation with a fixed number of intervals and a timeout period. According to [1], the recommended configuration settings for heartbeat protocols are limiting time interval to 150 ms and 10 timeouts. This setting captures the network variability due to versatile network delays between the network devices and the instance pauses caused by maintenance actions and software updates. Therefore, the maximum period that elapses when the offloading site is considered to be unavailable is 1.5 seconds which implies that the response time failure cost is fixed to $\alpha_t = 1.5$ and task response time Equation (11) is updated into Equation (17). In the case that no offloading failures occur during the offloading process, then the time failure cost is $\alpha_t = 0$.

$$t_v(v,q_i,q_j) = t_u(v,q_i,q_j) + t_c(v,q_j) + t_d(v,q_j,q_i) + \alpha_t \tag{17}$$

However, regarding the energy consumption failure cost α_e, since it is measured from the mobile device perspective, we distinguish three separate use cases. First, when the task is offloaded from the remote infrastructure to the mobile device and failure occurs during the download data transmission. Second, when the task is offloaded from the mobile device to the remote infrastructure and the failure occurrs during the upload data transmission. And thirdly, when the task is offloaded between two remote offloading sites while the mobile device is hibernating in the idle operational state. All three cases are shown in Equation (18) respectively and they update the energy consumption model in Equation (19) that is

extended with an additional energy cost factor α_e.

$$\alpha_e = \begin{cases} \alpha_t \times p_d \\ \alpha_t \times p_u \\ \alpha_t \times p_{idle} \end{cases} \quad (18)$$

$$e_v(v,q) = t_c(v,q) \times p_c + t_u(v,q) \times p_u + t_d(v,q) \times p_d + \alpha_e \quad (19)$$

3.8 MDP-SVR Edge Offloading Algorithm

The Algorithm 1 shows the MDP-SVR Edge offloading algorithm. The goal is to obtain reliable Edge offloading decisions through the SVR availability prediction of the offloading sites and efficient offloading to minimize both mobile device energy consumption and application response time. Coupling the offloading reliability with the performance efficiency boosts the latter. Additionally, the Edge offloading process is shown in Algorithm 2 which incorporates the MDP-SVR algorithm for obtaining the offloading decision policies and executing them accordingly in the runtime.

In the Algorithm 1, the for loop on Lines 4-9 iterates over the application tasks and computes the energy consumption and the response time assuming it is executed on each offloading site and stores in arrays on Lines 8 and 9. This is used later by the PIA algorithm that computes which offloading site is the best choice for offloading. In Line 12, the SVR algorithm predicts the offloading site availability based on the training dataset and forwards the estimation to Line 13 based on the computed probability matrix P. On Line 14, the MDP reward matrix R is computed and forwarded it together with other MDP parameters to the PIA algorithm that computes the offloading decision policy $\pi^*(s)$ (Line 15) and returns the output in Line 16. In Algorithm 2, the Edge Offloading Process is shown where task offloading is performed based on the output of the MDP-SVR algorithm given in Algorithm 1. In Lines 1-4 are the variables declaration and getting offloading decision policy $\pi^*(s)$ from the MDP-SVR algorithm. Then in the for loop (Lines 5-19), the task offloading is performed based on the offloading decision policy. If the task is offloaded on a site that experienced a failure during runtime then offloading is classified as failed (Lines 8) and the next best alternative should be considered (Line 12). The iteration is stopped until the alternative offloading is successful on site that does not experience the failure (Lines 15-16) or all possible solutions are exhausted due to failures or limited resources on the infrastructure (Line 10). The output result of the entire offloading process is a feasible offloading policy that succeeds to offload task (Line 20).

Algorithm 1 MDP-SVR Algorithm

```
1:  procedure MDP_SVR_ALGO(S, A, train_dataset, tasks)
2:      energy_vector ← array() ▷ Store energy consumption for each offloading sites
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, s_0)   ▷ PIA algorithm returns offloading decision policy
16:     return < π*, Q >
17: end procedure
```

Algorithm 2 Edge Offloading Process

```
1:  procedure OFFLOADING_PROCESS(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 > ← MDP_SVR_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 failure occurs then another a action should be considered
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
```

4 EVALUATION

4.1 Experimental Setup

We evaluate the proposed MDP-SVR framework under simulation conditions and compare it with other baseline ODE decision engines. The simulator is implemented in Python 3.6.5 on machine ThinkPad T470p machine that runs 64-bit Windows 10 OS with 16 Gb RAM, and dual-core i7-7700HQ CPU of 2.80 GHz and 2.81 GHz. We provide the simulator available to other researchers online [1].

4.1.1 Simulation scenario. The workload chain of mobile applications initiates on the mobile device and terminates when all tasks are executed. During the application runtime, ODE computes the offloading decision policy and offloads tasks accordingly. The offloading failure can occur during the runtime on the server or network links and failure costs are computed accordingly. Additionally, the assumption is that the ODE performs only the data offloading but not the computation offloading. This is justifiable

[1] https://github.com/jzilic91/edge/tree/master/MDP-SVR

since the computation offloading can raise security and privacy issues. For example, offloading non-verified and malicious snippets of executable code can damage end-users and service providers, economically and legally. Instead, the computation part of the mobile application is replicated on all offloading sites as a macro-service. Since the system architecture that we use in the simulation model is limited in size, this does not represent a system bottleneck.

4.1.2 Offloading sites. The simulation model includes five offloading sites which consist of a single mobile device, a single Cloud data center, and three Edge servers where each is a different type as explained in 3.1. The offloading site resources are limited and assumed static during the runtime. The hardware and network specifications are described in Tables 1 and 2 similar to [39] which suits our needs since it expresses the magnitude of the hardware and network capabilities ratio between the complementary parts of the infrastructure appropriately. The hardware and network capabilities that are included in the simulation model are CPU computation power, RAM capacity, data storage size, network latency, and network bandwidth rate. The entire infrastructure follows the mesh network topology which implies that each offloading site has assured at least one network link towards all other offloading sites. This topology is considered to be robust against failures and reduces maintenance costs [6]. The wireless network links of the mobile device are assumed to follow IEEE 802.11 wireless network speeds, while network links between Edge and Cloud are fixed based on already established Ethernet network standards such as Fast Ethernet (100 Mbps) and 1GBit Ethernet links (987 Mbps). This implies that remote infrastructure together with a mobile device is localized at a single geographical location. Notice that connections with Cloud also exhibit the Internet latency distribution $\varphi(\mu, \sigma)$ due to the transmission delay which varies between 100 and 300 ms according to [15]. To obtain the latency values between the aforementioned range, we employ a Gaussian distribution with mean μ = 200 and standard deviation σ= 33.5 ms similar to [13].

Table 1: Hardware specifications

Node	CPU (GHz)	RAM (GB)	Storage (GB)
Edge database server	5	8	500
Edge computational server	8	8	250
Edge regular server	5	8	250
Cloud data center	12	128	1000
Mobile device	1	8	16

Table 2: Network specifications

Links		Latency (ms)	Bandwidth (Mbps)
Mobile	Edge	15	5.5/20
Mobile	Cloud	54 + $\varphi(\mu, \sigma)$	20
Edge	Cloud	15 + $\varphi(\mu, \sigma)$	100/987
Edge	Edge	10	100/987

4.1.3 Mobile applications. The mobile applications that will be used for the Edge offloading evaluation are (i) *Facebook* as a use case scenario of posting pictures on Facebook, (ii) *GPS navigation* that navigates the traffic drivers to their destination, (iii) *Facerecognizer*, as the image processing application which recognizes facial structures in the images, (iv) *Antivirus* that scans the software and compares potential software virus signatures with the registered ones in the database, and (v) *Chess*, as an interactive game where AI software agent tries to anticipate player chess moves. All five aforementioned mobile applications are sampled according to the probability distribution computed from the LiveLab application usage traces [27] as in previous work [14]: (i) *Facebook*: 45%, (ii) *GPS navigation*: 30%, (iii) *Facerecognizer*: 10%, (iv) *Antivirus*: 5%, (v) *Chess*: 10%. The probability is defined as the likelihood that the next application will be ready for offloading. Within the same application, tasks can be diverse in all segments. Table 3 shows the application task classification which are referenced in Tables 4, 5, 6, 7, and 8.

Table 3: Application task specifications

Type	CPU	Input data	Output data
DI	100-200 MI	30-40 KB	50-60 KB
CI	550-650 MI	4-8 KB	4-8 KB
Moderate	100-200 MI	4-8 KB	4-8 KB

Table 4: Facebook task specifications

Task	Type	RAM	Offloadable
FACEBOOK_GUI	Moderate	1 GB	False
GET_TOKEN	Moderate	1 GB	True
POST_REQUEST	Moderate	2 GB	True
PROCESS_RESPONSE	Moderate	2 GB	True
FILE_UPLOAD	DI	2 GB	False
APPLY_FILTER	DI	2 GB	True
FACEBOOK_POST	DI	2 GB	False
OUTPUT	Moderate	1 GB	False

Table 5: GPS Navigator task specifications

Task	Type	RAM	Offloadable
CONF_PANEL	Moderate	1 GB	False
GPS	Moderate	3 GB	False
CONTROL	CI	5 GB	True
MAPS	DI	5 GB	True
PATH_CALC	DI	2 GB	True
TRAFFIC	DI	1 GB	True
VOICE_SYNTH	Moderate	1 GB	False
GUI	Moderate	1 GB	False
SPEED_TRAP	Moderate	1 GB	False

Table 6: Facercognizer task specifications

Task	Type	RAM	Offloadable
GUI	DI	1 GB	False
FIND_MATCH	DI	1 GB	True
INIT	DI	1 GB	True
DETECT_FACE	DI	1 GB	True
OUTPUT	DI	1 GB	False

Table 7: Antivirus task specifications

Task	Type	RAM	Offloadable
GUI	Moderate	1 GB	False
LOAD_LIBRARY	DI	1 GB	True
SCAN_FILE	DI	2 GB	True
COMPARE	DI	1 GB	True
OUTPUT	Moderate	1 GB	False

Table 8: Chess task specifications

Task	Type	RAM	Offloadable
GUI	Moderate	1 GB	No
UPDATE_CHESS	Moderate	1 GB	Yes
COMPUTE_MOVE	CI	2 GB	Yes
OUTPUT	Moderate	1 GB	No

4.1.4 Failure dataset. Currently, there does not exist a real-world Edge Computing failure dataset that is available for scientific research due to the novelty of technology and equipment accessibility. Consequently, we adopt failure traces from a general distributed computing infrastructure to the Edge Computing scenario. The dataset is made publicly available by Los Alamos National Laboratory (LANL) [26]. Although the LANL dataset is not collected on an Edge infrastructure, it possesses certain properties that suit our scenario such as a large number of computational nodes, distributed geographical locations, and heterogeneous hardware characteristics. The LANL dataset contains around 23,000 failure traces recorded on 22 different systems at the LANL site. It covers 4,750 nodes with 24,101 processor units with the life span from 1996 to 2005. Schroeder and Gibson [26] provide a detailed list of hardware characteristics for each of 22 systems with computation and data storage capabilities. In our work, these real-world failures are incorporated into the simulation model as failures at offloading sites. We classify them based on the aforementioned hardware characteristics. Since our focus is estimating offloading site availability, we compute availability distribution A_t according to Equations, (4) (5), (6), and (7) instead of directly parsing failure times. Training and testing data points are sampled from availability distribution $A(t)$ that is computed from different nodes. Each data point is interpreted as a probability that the observed offloading site will be in an operational and committable state when the task is offloaded on that site for execution.

Computing the availability for the entire LANL dataset is not computationally feasible due to a wide lifespan and a large number of nodes. Moreover, aggregating availability distributions over all nodes likely results in a randomized distribution that hardly exhibits any kind of pattern. Instead, we pick several nodes from the dataset to compute availability distributions similar to previous work [8, 30]. The overview of selected nodes from the LANL dataset is shown in Table 9. The nodes are selected according to their availability levels categorized as *low*, *medium* and *high* relative to other nodes in the same dataset configuration. Additionally, *high volatile* node is also present which presents a node that is highly available but exhibits a larger variance in the availability sampling distribution due to a few severe failures which are observed as an outlier.

The resources are named <*systemID_nodenumber*> where both index numbers are obtained from the original dataset. The selected nodes, together with the aforementioned availability levels, are categorized according to the hardware characteristics as explained in Subsection 3.1. The nodes from systems 5 and 7 are most suitable to the Edge database (ED) category due to a large number of nodes (larger data storage). The edge computational (EC) availability distribution is sampled from nodes of systems 19 and 20 which have a higher ratio of processors per node (higher computational power). Edge regular (ER) is sampled from 3, 4, and 16 systems due to lower processor per node ratio, a minimum quantity of network interface cards, and a moderate number of nodes compared relatively to the Edge database (ED) and the Edge computational (EC). The Cloud category is sampled only from 22_0 node since this single node has the highest processor per node ratio and RAM capacity in the entire dataset, which represents resource abundance in the Cloud data centers (CD).

4.2 Simulation Results

4.2.1 Baseline decision engines and confidence intervals. For the performance comparison, we introduce two baseline ODE engines from the state-of-the-art solutions in literature, (i) *EFPO* [39] handles the offloading mechanism through the MDP process with a probabilistic reliability model based on the general bathtub curve for the repairable systems from the reliability engineering [28], and (ii) *Energy Efficient* (EE) refers also to the MDP control offloading process but without the failure predictability feature. In the simulation evaluation, we chain 40 mobile applications into a single workload execution that is executed successively and repeated 10,000 times to gain the statistical significance and the result validity within the 95% confidence interval. In all related response time results, the confidence interval is in the worst-case ± 0.082 s while for the energy consumption is ± 0.073 J. For failure rates, the confidence interval in the worst case is up to ± 0.0711 failures. As a worst-case, we refer to the dataset configuration that exhibits highest result deviation. The workload chain of 40 mobile applications is selected appropriately as a middle-ground due to the simulation high time-consumption, which can last more than a day, and a sufficient number of various application executions for statistical significance.

4.2.2 Offloading results. Figures 4 and 5 illustrates the mobile application response time and the mobile device energy consumption for the workload chain of 40 mobile applications. In both figures, the MDP-SVR offloading algorithm outperforms both EFPO and EE decision engines in all 5 dataset configurations. Both response time and energy consumption results are similar due to the linear relationship between the both models that are explained in Section 3.5

Table 9: Dataset configurations

Offloading Site	Dataset configurations				
	Low availability EC (DC1)	**Medium availability EC (DC2)**	**Medium availability EC and ER (DC3)**	**High volatility ER (DC4)**	**High availability Edge (DC5)**
Edge database (ED)	High availability ED (node: 7_1)	High availability ED (node: 5_158)	High availability ED (node: 5_165)	High availability (node: 5_243)	High availability ED (node: 5_48)
Edge computational (EC)	Low availability EC (node: 19_1)	Medium availability EC (node: 19_11)	Medium availability EC (node: 19_4)	High availability EC (node: 19_8)	High availability EC (node: 20_41)
Edge regular (ER)	High availability ER (node: 3_0)	High availability ER (node: 16_80)	Medium availability ER (node: 4_55)	High volatility ER (node: 4_1)	High availability ER (node: 4_3)
Cloud (CD)	Cloud (node: 22_0)	Cloud (node: 22_0)	Cloud (node: 22_0)	Cloud (node: 22_0)	Cloud (node: 22_0)

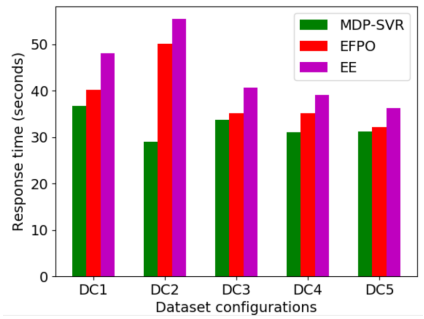


Figure 4: Application response time

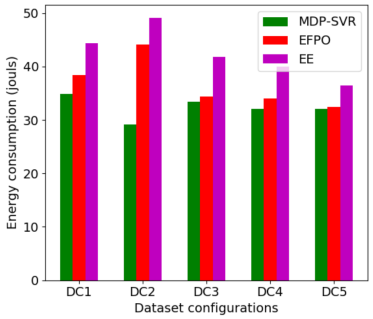


Figure 5: Mobile energy consumption

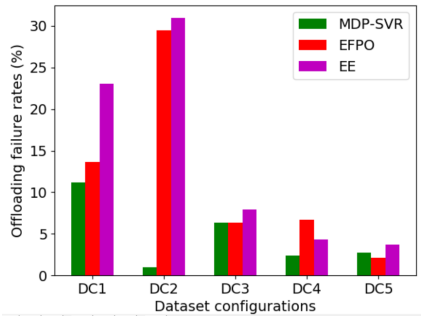


Figure 6: Offloading failure rates

and 3.6 in detail. In most cases, the EE engine has the worst performance due to a lack of failure predictability features. This causes a larger quantity of observed costly offloading failures that rises to 30% (Figure 6). Additionally, the significant amount of tasks are offloaded on the remote Cloud (up to 22.27% in Figure 7d) which is much more then other two alternatives (between 0.32% in Figure 7c and 3.38% in Figure 7a). Consequently, this baseline exhibits worse performance due to larger network latency and lower bandwidth rate between the mobile device and the Cloud. EFPO, on the other hand, yields moderate performance but worse comparing to our MDP-SVR solution since the general reliability bathtub concept is in practice rarely replicated [28]. The MDP-SVR relies on the historical training data which gives more accurate and reliable information about the specific offloading site failure behavior. For instance, in dataset configuration 2, the MDP-SVR is absolutely and relatively superior than in other dataset configurations since the failure rate is around 1% (Figure 6). This dataset configuration is specific since it exhibits the highest quantity of failures of all configurations, especially on the Edge Computational server which the MDP-SVR in Figure 7b estimated as an unreliable offloading site. Only 3.21% of tasks are offloaded on the aforementioned site.

The prediction capabilities of the MDP-SVR algorithm are proved to be a viable solution that can handle the availability distributions that exhibit more volatile behavior. EFPO and EE, due to the absence of a qualitative prediction engine, suffered the highest failure rates up to almost 30% as seen in Figure 6 in the dataset configuration 2. On the other hand, dataset configuration 5 exhibits the smallest number of failures, and consequently, the time and energy performance of all three solutions are closer than in other dataset configurations, which is between 33 and 36 seconds and 33 and 36 joules respectively. When the amount of failures is smaller then all three solutions exhibit similar performances since all depend on the same MDP control offloading algorithm. However, regarding the offloading reliability, in the same dataset configuration, the MDP-SVR did not manage to outperform the EFPO algorithm where the failure rate is up to low 1.5% (Figure 6). The reason lies in the parameter selection of the SVR parameters mentioned in Section 3.2. Although parameter selection can handle outliers that are specific to the availability distributions, its near-optimal performance cannot handle all cases, specifically when a smaller quantity of failures is observed.

Regarding the offloading distributions, there are several insights worthy of mentioning. The EE ODE engine in 3 out of 5 dataset configurations (Figures 7b, 7d, 7e), ignores the Edge Regular offloading site completely since it is resourcefully inferior to the other offloading sites. The MDP-SVR discriminates Edge Regular even more since combined with the availability data it becomes even less favorable due to higher failure rates on the offloading site. Moreover, MDP-SVR also ignores the Cloud data center offloading site. This is due to several factors. First, unfavorable network characteristics yields extended task offloading. Second, using the same node 22_0 during the entire experiment yields static and reliable results. And lastly, from the availability perspective, the Cloud is perceived as an moderately available offloading site. Although the Cloud should be more available site due to redundant hardware and network connections, the real-world node 22_0, which represents the Cloud

in the simulation model, is integrated into the LANL system in the last year of the measurement. This implies that a significant part of the node's lifespan was in the 'infant mortality' phase where failure rates are higher due to software upgrades, installation errors, and mishandling. The MDP-SVR always finds at least one available Edge offloading site with sufficient resources that yields a faster and more reliable solution. EFPO, on the other hand, does not discriminate particular offloading sites as strong as the aforementioned decision engines. The reason lies in the bathtub reliability curve which depends on the reliability parameter MTBF (mean-time-between-failures) which does not capturing the complexity of the failure behavior during the lifespan. It is assumed to function in the general situation where the system is operating in the so-called 'useful life period' where failure rates are low, constant, and appearing randomly. This ODE can yield higher performance and reliability than the case without the failure predictability feature as the EE but not higher accuracy enough to overperform MDP-SVR.

5 LEARNING COMPLEXITY ANALYSIS

The main goal of the learning complexity analysis is to estimate the computational overhead of the SVR and the MDP algorithms used in the offloading site availability estimation and the optimal offloading approximation. In the context of limited Edge resources and expansive ML computational overhead, it is vital to determine the feasibility of the proposed solution.

5.1 SVR PAC Analysis

The formal definition of an upper bound on the sample complexity for the SVR is given by [16]:

$$m \leq \frac{1}{\epsilon(1-\sqrt{\epsilon})}\left(2VC(H)\ln\frac{6}{\epsilon}+\ln\frac{2}{\delta}\right), \forall 0<\delta<1 \qquad (20)$$

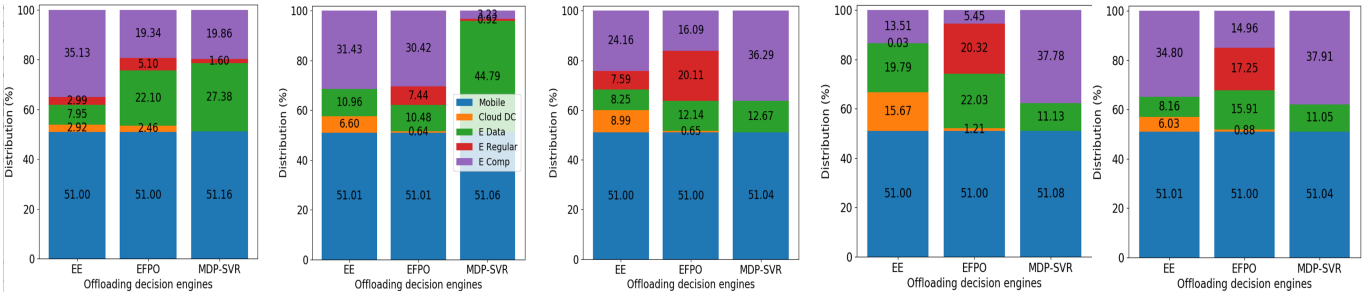
where m is the training sample size, ϵ is error accuracy, σ is confidence level and $VC(H)$ is the VC dimension of the hypothesis space H. The Equation (20) computes the sample complexity for hypothesis spaces H that exhibit on real-valued attributes in the training data. This fits our SVR prediction model for the offloading site availability distribution. The $VC(H)$ is the only parameter from Equation (20) that is not explicitly user-defined and it depends on the used kernel model. The VC dimension for the linear SVM models is $VC(H) = d + 1$ where d represents the data dimensionality [34]. For instance, if the dataset is defined in the 2-dimensional space ($d = 2$), then $VC(H) = 3$. The interpretation is that the linear SVM model can shatter a maximum of 3 data points, which are defined in the 2-dimensional space. Similar for the polynomial SVM models where the VC dimension is defined as $VC(H) = (\frac{d+p-1}{p})$ with p as the degree of the used polynomial model [9]. The third option for an SVM model is a Gaussian model so-called Radial Basis Function (RBF). This model has been proven to have better prediction performance than other aforementioned models [29]. However, in work [9] it is mentioned that due to the efficiency and complexity of RBF kernel, it can shatter an infinite number of data points which implies that the VC dimension is $VC(H) = \infty$. Importing this value into the Equation (20), the upper bound is eliminated since the number of training samples goes to ∞. While this problem may not be formally PAC learnable, the work of [33] shows that theoretical considerations are overly pessimistic in a practical setting. Based on this finding, the SVR empirically can have a good performance despite the theoretical estimation of the sample complexity. Although in a theoretical setting it is assumed that the data is identically and independently distributed, in practice this is usually not the case. This allows our model to learn from patterns observed in real-world data even in the absence of theoretical guarantees. Consequently, the SVR algorithm never reaches its full capacity which produces the discrepancy between the theoretical and the empirical results. We will show this in Subsection 5.2.

The computational complexity is the second characteristic of the SVR algorithm that we want to take into consideration. Research [2] showed that the computational complexity of the SVM is $O(m^3)$ where m represents the training sample size. This complexity is considered to be computationally intensive as long the training sample size is large. Its effect on time consumption of the SVR algorithm will be empirically verified in Section 5.2. The author of the aforementioned work performed the complexity analysis upon the LibSVM library on which our SVR model is based on using the wrapper Python library *sklearn.svm*.

5.2 Empirical Results of SVR Learning Complexity Analysis

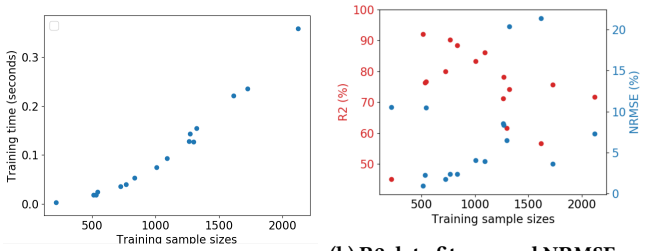
There is a discrepancy between the theoretical and the empirical results due to the agnostic distribution-free setting in theory and the real-world data distribution in practice. The empirical SVR training time measurements together with the training sample sizes are presented in Figure 8a. The SVR training time strongly depends on the training sample size and exhibits a poly-like appearance as analyzed in Section 5.1. Each data point is measured separately from the aforementioned nodes from Table 9. With finite but large training sample sizes, the SVR training time converges in a fast period. The maximum training time is 0.389 s with a sample size of 2124 data points from the node 3_0. While predictions in related works [12, 18] are generated incrementally in an online fashion, our employed algorithm instead enables us to create predictions for the whole test set in a 4-to-1 (80%-20%) training-test setup, which avoids continuous algorithm execution and resource consumption. For further execution, it can be extended to re-trigger periodically to keep predictions up-to-date or measuring an error accuracy continuously and triggering SVR once accuracy goes below some defined threshold. Moreover, the model quality measures should also be applied to evaluate the performances of the SVR and the impact of the sample complexity.

Figure 8b illustrates more random-like distribution of the dependency between R2 score and NRMSE (normalized root mean square error) on the one hand and training sample sizes on the other hand. For a reminder, each data point is measured from separate nodes that are shown in Table 9. Each node has a different training sample size due to different life span and exhibits different availability distribution. The results show that predicting data from separate nodes yields different results since determining the SVR hyperparameters is data-dependent (Section 3.2). This acknowledges that the estimation of the sample complexity should be based on the actual data distributions and not agnostically obtained (an assumption that data distribution is i.i.d.). Thus, the SVR yields different prediction results and consequently, it cannot reach its maximum capacity



(a) DC1 (b) DC2 (c) DC3 (d) DC4 (e) DC5

Figure 7: Offloading distribution with different dataset configurations



(a) SVR training time

(b) R2 data fitness and NRMSE error accuracy of SVR

Figure 8: SVR empirical measurements

measured as the VC dimension. While SVR is overall well suited for our problem, it is not a robust algorithm when an excessive amount of outliers are present in the data [17] as some R2 score result does not even cross 50% of the data fitness in Figure 8b.

5.3 Markov Decision Process Complexity Analysis

The MDP control offloading algorithm has a computational complexity of $O(|S| * |A|)$ per task offloading. The computational complexity does not reflect the complexity of the model-checking algorithm (VIA or PIA). Since the MDP algorithm is a computationally expensive operation for mobile devices, we alternatively propose placing the MDP algorithm on a remote dedicated server to produce the offloading decision policy. The remote server computes the offloading decision policy based on the MDP model information from the mobile device and returns the offloading decision policy in the matrix form. If the user frequently consumes the same applications and remote infrastructure resource does not change during runtime, then different offloading decision policies can be cached on the mobile device. In the case of the intermittent or failed connection between the mobile and the remote server, a mobile device detaches from the server and initiates the MDP algorithm with the last stored MDP model information. It is possible and sufficient since MDP exhibits the so-called Markov Property where the future states depend only on the present state, not the sequence of states that preceded it. When a remote server is back online again, then the MDP algorithm on the mobile device switches off and forwards the current model information to continue the MDP execution on the remote server.

6 RELATED WORK

The offloading concept is considered to be a viable solution to tackle and overcome the hardware limitations of mobile devices and accelerate the application response time. Relevant offloading frameworks that appear in the Mobile Cloud Computing (MCC) literatue is summarized in [3]. The frameworks are mostly concerned with the optimization objectives such as mobile device or infrastructure energy efficiency, simplifying code development of offloading applications, and reducing workloads but without considering offloading failures. Similar frameworks developments also appear in the Mobile Edge Computing (MEC) literature [22] where performance utilities as execution delay and energy consumption are minimized separately or jointly. The same narrative followed in the Edge Computing field where offloading frameworks employ multi-objective optimization algorithms [13, 14] to optimize application response time, mobile device battery lifetime, users' cost, and providers' profit. No offloading failures are considered in their model. Researchers that considered offloading failures in the offloading systems used M/M/1 queue model [36], checkpointing mechanism [24], local re-execution and timeout mechanism [35] and recovery mechanisms [25]. All of them are using reactive maintenance approaches and not adapted for the Edge offloading.

Applying discrete stochastic optimization algorithms such as MDP in the offloading optimization is not unprecedented. The work presented in [31] modeled the Mobile Cloud offloading control process as an MDP with considering the stochastic wireless channels between the Cloud and the mobile device. However, it does not consider offloading failures. A similar approach is applied to Edge Computing by [5] but with the same aforementioned limitation. [38] applied the MDP to optimize the offloading by considering offloading failures introduced by the intermittent connections between the Cloudlets and the mobile device. This solution is not adapted for Edge Computing settings and considers only link failures. Our previous work [39] did consider offloading failures in Edge offloading process by applying the MDP approach but based on a simpler and more general bathtub reliability model that is hardly replicated in practice. Researchers in [8] employed Bayesian Networks to estimate the future availability of virtual machines

on Edge data centers to minimize Service Level Objective violations. However, this work is not adapted for offloading and QoS parameters are channeled through availability objective. There are also another ML approaches applied to handle reliability issues in the distributed computing system such as applying artificial neural network [37] and support vector machine [23] but none of them is adapted for Edge Computing and offloading settings. In this work, we ensure this adaptation by MDP and SVR algorithms to handle nondeterministic and stochastic offloading control process, combined with offloading site availability prediction to boost system performance and reliability.

7 CONCLUSION AND FUTURE WORK

Proposed Edge offloading framework is based on the MDP formal framework where the offloading sites were represented as states, offloading decisions as actions, offloading site availability as transition probability, and application response time and energy consumption as reward functions. The offloading site availability is computed via the SVR algorithm based on the failure trace logs from the local sites and forwarded it to the MDP control algorithm that computes the offloading decision policy. The mobile applications are modeled as DAGs, fragmented into smaller tasks that are offloaded on the remote offloading site, instead of the entire application. The evaluation is made in a simulation environment due to a lack of Edge equipment. The failure traces originate from the LANL site for an HPC system but can be reused in the Edge Computing environment due to shared characteristics. The mobile applications are sampled from the LiveLab application usage traces. The MDP-SVR framework shows improved results compared with other baseline offloading algorithms regarding overall performance and offloading reliability. This can give potential and be an inspiration to the continuation of developing and researching new Edge offloading solutions that take into account the reliability of the system to mitigate failures. As future work, we will focus on replication strategies in the Edge Computing environment for efficient and resilient resource provisioning.

ACKNOWLEDGEMENTS

This work is funded through the 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] Abdiansah Abdiansah and Retantyo Wardoyo. 2015. Time complexity analysis of support vector machines (SVM) in LibSVM. *International journal computer and application* 128, 3 (2015), 28–34.
[3] Khadija Akherfi, Micheal Gerndt, and Hamid Harroud. 2018. Mobile cloud computing for computation offloading. *Applied Comp. and Inf.* 14, 1 (2018), 1–16.
[4] Khalid Alasmari, R.C. Green, and Mansoor Alam. 2018. Mobile Edge Offloading Using Markov Decision Processes. 80–90. https://doi.org/10.1007/978-3-319-94340-4_6
[5] Khalid R Alasmari, Robert C Green, and Mansoor Alam. 2018. Mobile edge offloading using MDP. In *Int'l. Conf. on Edge Computing*. Springer, 80–90.
[6] Naomi Alpern. 2009. *Eleventh Hour Network+: Exam N10-004 Study Guide*. Syngress.
[7] Zeeshan Amin, Harshpreet Singh, and Nisha Sethi. 2015. Review on fault tolerance techniques in cloud computing. *International Journal of Computer Applications* 116, 18 (2015).
[8] Atakan Aral and Ivona Brandic. 2017. Quality of service channelling for latency sensitive edge applications. In *2017 IEEE International Conference on Edge Computing (EDGE)*. IEEE, 166–173.
[9] Christopher JC Burges. 1998. A tutorial on support vector machines for pattern recognition. *Data mining and knowledge discovery* 2, 2 (1998), 121–167.
[10] Lijuan Cao and Qingming Gu. 2002. Dynamic support vector machines for non-stationary time series forecasting. *Intelligent Data Analysis* 6, 1 (2002), 67–83.
[11] Vladimir Cherkassky and Yunqian Ma. 2004. Practical selection of SVM parameters and noise estimation for SVM regression. *Neural networks* 17, 1 (2004), 113–126.
[12] 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.
[13] Vincenzo De Maio and Ivona Brandic. 2018. First hop mobile offloading of dag computations. In *IEEE/ACM Int'l. Symp. on Cluster, Cloud and Grid Comp.* 83–92.
[14] Vincenzo De Maio and Ivona Brandic. 2019. Multi-Objective Mobile Edge Provisioning in Small Cell Clouds. In *ACM/SPEC Int'l. Conf. on Perf. Eng.* 127–138.
[15] Mark DeVirgilio, W David Pan, et al. 2013. Internet delay statistics: Measuring internet feel using a dichotomous hurst parameter. In *IEEE Southeastcon*. 1–6.
[16] David Haussler. 1990. *Probably approximately correct learning*. University of California, Santa Cruz, Computer Research Laboratory.
[17] Josh Hoak. 2010. The Effects of Outliers on Support Vector Machines. *Portland State University* (2010).
[18] Cong Jin and Shu-Wei Jin. 2014. Software reliability prediction model based on support vector regression with improved estimation of distribution algorithms. *Applied Soft Computing* 15 (2014), 113–120.
[19] Karthik Kumar and Yung-Hsiang Lu. 2010. Cloud computing for mobile users: Can offloading computation save energy? *Computer* 4 (2010), 51–56.
[20] Nicholas Littlestone. 1990. Mistake bounds and logarithmic linear-threshold learning algorithms. (1990).
[21] Nick Littlestone, Manfred K Warmuth, et al. 1989. *The weighted majority algorithm*. University of California, Santa Cruz, Computer Research Laboratory.
[22] 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.
[23] 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.
[24] 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 *IEEE Int'l. Conf. on Communications*. 1856–1860.
[25] Dimas Satria, Daihee Park, and Minho Jo. 2017. Recovery for overloaded mobile edge computing. *Future Generation Computer Systems* 70 (2017), 138–147.
[26] 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.
[27] Clayton Shepard, Ahmad Rahmati, Chad Tossell, Lin Zhong, and Phillip Kortum. 2011. LiveLab: measuring wireless networks and smartphone users in the field. *ACM SIGMETRICS Performance Evaluation Review* 38, 3 (2011), 15–20.
[28] Anthony M Smith. 1993. *Reliability-centered maintenance*. McGraw-Hill New York.
[29] Alex J Smola and Bernhard Schölkopf. 1998. *Learning with kernels*. Vol. 4. Citeseer.
[30] Yongmin Tan, Hiep Nguyen, Zhiming Shen, Xiaohui Gu, Chitra Venkatramani, and Deepak Rajan. 2012. Prepare: Predictive performance anomaly prevention for virtualized cloud systems. In *IEEE 32nd International Conference on Distributed Computing Systems*. IEEE, 285–294.
[31] Mati B Terefe, Heezin Lee, et al. 2016. Energy-efficient multisite offloading policy using MDP for MCC. *Pervasive and Mobile Computing* 27 (2016), 75–89.
[32] Vladimir Vapnik. 2013. *The nature of statistical learning theory*. Springer science & business media.
[33] Vladimir Vapnik, Esther Levin, and Yann Le Cun. 1994. Measuring the VC-dimension of a learning machine. *Neural computation* 6, 5 (1994), 851–876.
[34] Vladimir N Vapnik. 1999. An overview of statistical learning theory. *IEEE transactions on neural networks* 10, 5 (1999), 988–999.
[35] Qiushi Wang, Huaming Wu, and Katinka Wolter. 2013. Model-based performance analysis of local re-execution scheme in offloading system. In *IEEE/IFIP International Conference on Dependable Systems and Networks*. 1–6.
[36] Huaming Wu. 2018. Performance modeling of delayed offloading in mobile wireless environments with failures. *IEEE Communications Letters* 22, 11 (2018), 2334–2337.
[37] Kai Xu, Min Xie, Loon Ching Tang, and SL Ho. 2003. Application of neural networks in forecasting engine systems reliability. *Applied Soft Computing* 2, 4 (2003), 255–268.
[38] Yang Zhang, Dusit Niyato, and Ping Wang. 2015. Offloading in mobile cloudlet systems with intermittent connectivity. *IEEE Transactions on Mobile Computing* 14, 12 (2015), 2516–2529.
[39] Josip Zilic, Atakan Aral, and Ivona Brandic. 2019. EFPO: Energy Efficient and Failure Predictive Edge Offloading. In *Proceedings of the 12th IEEE/ACM International Conference on Utility and Cloud Computing*. 165–175.