Win with What You Have: QoS-Consistent Edge Services with Unreliable and Dynamic Resources

Zheng Song and Eli Tilevich
Software Innovations Lab, Dept. of Computer Science, Virginia Tech
Email: {songz, tilevich}@vt.edu

***Abstract*—Mobile and energy harvesting devices increasingly provide resources for edge environments. These devices' mobility and limited energy budgets may cause failures and poor performance. The reliability and efficiency of edge services can be improved with equivalent microservices that satisfy application requirements by different means: execute equivalent microservices in the predefined patterns of fail-over to minimize execution costs or speculative parallelism to reduce latency. However, given the vast dissimilarities in resource availability and capability across edge environments, being limited to these predefined patterns when implementing edge services causes inconsistent QoS. To address this problem, we provide QoS-consistent edge services by customizing the execution of equivalent microservices. Our system estimates the environment-specific QoS of equivalent microservices and dynamically generates execution strategies that best satisfy given QoS requirements. We evaluate the effectiveness and performance of our system via simulations and benchmarks with realistic edge deployments. Our approach consistently outperforms the predefined execution patterns in satisfying the QoS requirements in unreliable and dynamic edge environments.**

***Index Terms*—Edge Services, QoS Optimization, Orchestration**

I. Introduction

Edge computing coordinates sensing, computation, and data storage resources at the edge of the network [30]. Being within the direct communication range of each other and the client, resource-providing edge devices offer the communication latency lower than that of cloud-based servers [11]. One way to expose edge-based resources to application programmers is via the service-oriented architecture (SOA). A service coordinates the execution of edge-based distributed tasks, implemented as edge microservices [31].

When it comes to provisioning services, cloud-based systems coordinate the execution of abundant and reliable resources. In contrast, *edge-based systems coordinate the execution of unreliable and dynamic resources*. The execution failure ratio of edge services tends to be higher than that of cloud services [2], [21], as it is often mobile or energy-harvesting [12], [24], [36] devices that supply edge resources. In edge environments, an execution can fail for multiple reasons: a mobile device moves out of communication range; an energy harvesting device becomes temporally unavailable, driven into sleep mode; a speech recognizer fails due to noise. Besides, cloud service vendors can always cost-efficiently allocate the required amount of pre-deployed resources, while edge services may need to be provided in diverse edge environments with dissimilar and often scarce resources [4].

To improve reliability, the state of the practice for cloud systems is to deploy replicated services on redundant cloud resources. On the contrary, edge systems rely on resource-scarce edge devices, rendering the replication solution inapplicable. Considering the wide range of sensors and data processing methods at the edge, our previous work MOLE [31] takes advantage of equivalent microservices, which provide the same functionality by different means and rely on dissimilar resources (e.g., (1) camera/image analysis, (2) motion sensors, and (3) wireless signal, used in place of each other for indoor localization [10]). These equivalent microservices can be executed in the fail-over pattern to improve reliability with minimal costs or in the speculative parallel pattern to improve reliability with minimal latency; we call such patterns *execution strategies*.

However, MOLE cannot always deliver QoS-consistent edge services, as it follows the specified fixed execution strategy across edge environments with vastly dissimilar resources. The resource dissimilarity across different environments yields constituent equivalent microservices with uncertain QoS, which in turn results in edge services that execute these microservice in predefined patterns delivering unpredictable and inconsistent QoS to the client. The state of the art lacks a frame of reference for identifying and expressing highly customized strategies for executing equivalent microservices, whose QoS performance can be estimated accurately.

In the approach presented herein, we provide reliable and QoS-consistent edge services with unreliable and dynamic resources. In particular, rather than follow predefined execution strategies (as in MOLE), we provide highly customized execution strategies that increase the QoS-consistency of edge services across edge environments. Our system employs a feedback loop [5] to monitor the environment-specific performance of edge microservices and dynamically generate execution strategies based on the service's QoS requirements.

The insight that motivates our system design is the dissimilar QoS of executing equivalent microservices by different strategies. To be able to generate a customized execution strategy that best fits the QoS requirements in a given edge environment, we explore the following system design questions: 1) how to express customized execution strategies; 2) how to determine all possible strategies for a given set of equivalent microservices; and 3) how to estimate the QoS of a strategy.

The contribution of this paper is threefold:

- **System Design**: We introduce an edge system design that provides reliable services with consistent QoS. Our design features a feedback loop that collects the environment-specific performance of microservices, as well as a generator that customizes execution strategies to best satisfy services' QoS requirements. To the best of our knowledge, this paper is the first to identify, define, and solve the problem of providing QoS-consistent services in dissimilar edge environments with dynamic resources.

- **Customized Execution Strategies**: We explore how to customize execution strategies of equivalent functionalities to best satisfy given QoS requirements. To the best of our knowledge, we are the first to be able to 1) formulate any customized execution strategy for equivalent functionalities; 2) determine what all possible execution strategies for any number of equivalent functionalities are and estimate their QoS.
- **Evaluation**: We systematically evaluate the efficiency and scalability of our system design as well as its actual performance by benchmarking edge services deployed and executed in real execution environments.

The rest of the paper is structured as follows: Section II discusses the obstacles of provisioning edge services. Section III presents our approach that determines all possible execution strategies for given equivalent microservices and estimates their QoS. Section IV gives an overview of our system design and execution strategy generation algorithm. Section V describes our evaluation results. Section VI compares our solution with the state of the art, and Section VII concludes the paper.

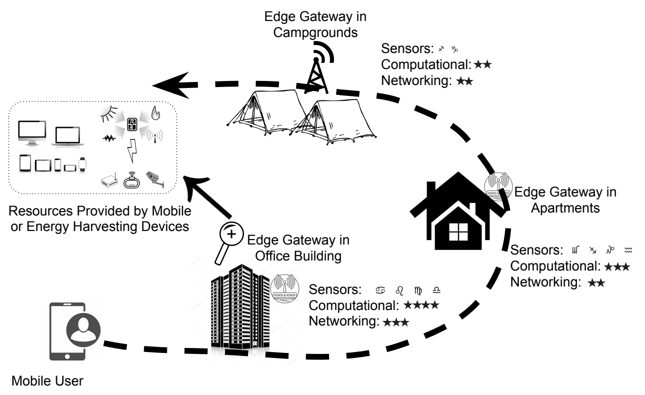
II. Problems in Provisioning Edge Services

By embracing the service-oriented architecture (SOA), edge executions across heterogeneous distributed devices are exposed as service invocations, thus shielding application developers from the necessity to implement low-level, platform-specific functionalities and D2D communication. Although SOA has become an industry standard for cloud computing [32], edge computing operates in fundamentally different execution environments, rendering cloud-based SOA designs inapplicable. While to meet the service level agreements for cloud services, their vendors only need to appropriately configure the abundant computational and network resources, edge service providers often have scarce, unreliable, and dynamic resources at their disposal, with which to meet the QoS requirements. To demonstrate the problems that these realities of edge computing present, consider the following example.

A. Motivating Example: Detecting Fire

One of the key functionalities of personal mobile assistants is to keep their owners safe. Such assistants can have a feature that periodically checks for the potential presence of fire to be able to alert its users and guide them to an escape route. To detect the presence of fire in the surrounding environment, the edge service `detectFire` can be queried in dissimilar environments that can range from office buildings to apartments, shopping malls, and even campgrounds. This service must be reliable, responsive, and cost efficient.

What hinders the QoS-guaranteed provisioning of such a service in dissimilar edge environments is their unreliable and dynamic resources [1], [7], [8], [21], [28], [31]. Fig. 1 demonstrates how a mobile device queries edge services in edge environments with dissimilar resources:



Edge Gateway in Campgrounds
Sensors:
Computational: ★★
Networking: ★★
Resources Provided by Mobile or Energy Harvesting Devices
Edge Gateway in Apartments
Edge Gateway in Office Building
Sensors:
Computational: ★★★
Networking: ★★
Sensors:
Computational: ★★★★
Networking: ★★★
Mobile User

Fig. 1: Edge Services in Dissimilar Edge Environments

1) Edge resources can be provided by mobile devices [3], [26] or energy harvesting stationary devices [24], [36]. With multiple mobile devices in the vicinity, they can be organized into a computing ensemble that can execute demanding edge services [3], [26]. However, typically owned by individuals, mobile devices are hard to predict or control, as their owners can move away or use them at any time, thus causing service failures. Besides, IoT devices increasingly rely on the energy harvesting technology [19], which accumulates ambient recyclable energy, including solar radiation, wind, human motion energy, and WiFi signals. However, these devices can be operated only intermittently: energy may be unavailable to harvest, taking time to accumulate to allow execution [23]. As a result, when executed on such devices, microservices cannot guarantee satisfactory reliability.

2) Besides, different edge execution environments may possess resources with dissimilar capabilities and capacities. For example, an office building may have built-in flame sensors for detecting fire, while apartments may only have smoke detectors; an indoor environment may have high-performance edge servers for computationally intensive tasks, while an outdoor environment may only have a solar-powered Raspberry Pi with much lower computational power. The resource difference across edge environments causes the dissimilar availability and performance of edge-based microservices.

B. MOLE: Reliability-enhanced Edge Services

Our previous work, MOLE [31], demonstrates that equivalent functionalities can be executed to improve the reliability of edge computing. Edge computing environments feature a wide range of sensors and data processing methods, so an application requirement can be fulfilled in multiple equivalent ways. MOLE enables edge service developers to specify the execution strategies for equivalent microservices, which include the fail-over and speculative parallel strategies. The `fail-over` strategy first tries executing a microservice; if it is unavailable or disabled, the execution switches to a back-up microservice. The `speculative parallel` execution strategy spawns the execution of all microservices simultaneously, proceeding as soon as any of them returns successfully. With both strategies improving reliability, fail-over is cost-efficient and speculative parallel is latency-efficient.

In the aforementioned example, to improve its reliability, `detectFire` can execute the equivalent microservices that detect 1) smoke by a surveillance camera; 2) smoke by smoke sensors; 3) flame by flame sensors; 4) the change of CO/CO_2 level by gas sensors; 5) the temperature change by a temperature sensor. We assume that the output of any one of these microservices, rather than their fusion, can detect fire. When one microservice fails, MOLE switches to its equivalent pair. Even if one or more microservices are unreliable, the edge service's overall reliability can still be guaranteed.

However, the overall performance of MOLE-specified services differs across edge environments with predefined execution strategies. For example, assume the `detectFire` service is developed in an environment `A` with edge-based small-scale data centers providing the computational power. Considering the latency of each equivalent microservice is pretty low, the developer specifies the execution strategy as "fail-over" for better cost-efficiency. However, while being executed in a different edge environment `B` with a Raspberry Pi providing the computational power, the "fail-over" execution may lead to an extremely long latency which is unexpected. Hence, our solution extends MOLE's reliability enhancement by introducing a novel system design that uses a feedback loop to generate environment-tailored execution strategies.

C. Customizing Execution Strategies to Optimize QoS

Due to the proliferation of unreliable execution environments (e.g., edge, IoT, etc.), the problem of optimizing their QoS has come to the forefront of distributed system design. This problem is exacerbated by these environments being unable to take advantage of existing designs that rely on standard resource deployments. In the approach presented herein, we put forward a novel optimization methodology that customizes the execution strategies for equivalent microservices.

Several prior approaches make use of the combined execution of equivalent functionalities. To improve service responsiveness, several cloud service instances are deployed and executed simultaneously [13], [27]. To improve reliability, automatic Workarounds provide automatic fail over with equivalent functionalities [9]. The emergence of IoT and edge computing gives rise to distributed execution environments that feature a wide range of sensors and processing methods, thus greatly increasing the variety and number of equivalent functionalities. However, all these existing approaches can execute equivalent functionalities in predefined execution patterns. The state of the art lacks a frame of reference for identifying and expressing highly customized strategies for executing equivalent microservices, whose QoS performance can be estimated accurately. The exploding numbers of equivalent functionalities of the emerging distributed environments present an untapped potential for optimization QoS by fully exploiting their customized execution.

III. Execution Strategies for Equivalent Microservices

Consider the aforementioned example: we use a, b, c, d, e to denote the five equivalent microservices for `detectFire`. For example, possible execution strategies for five equivalent microservices (a, b, c, d, e) include, but are not limited to: 1) fail-over: execute a, b, c, d, e in turns if the previous microservice fails; 2) speculative parallel: execute a, b, c, d, e simultaneously, returning the first obtained result; 3) first execute a and b simultaneously; if any of them succeeds, return the results; otherwise, execute c, d, e simultaneously and return the first available result; 4) first execute a, then b and c simultaneously; if none of them succeed, execute d first then e. To generate execution strategies that best satisfy given QoS requirements, we need to 1) find all possible execution strategies and 2) compare their QoS.

A. Expressing an Execution Strategy

An execution strategy expresses the invocation sequence of a set of equivalent microservices, which can be short-circuited when any microservice succeeds. Fig. 2 gives the EBNF grammar of an execution strategy. We express an execution strategy (denoted as `es`) by a set of equivalent functions (denoted as `eqvFunc`, i.e., a, b, c...) and the operators connecting them into an expression. The binary operators $-$ and $*$ denote a sequential and a parallel execution, respectively, and the parentheses operators denote that the execution strategy inside a pair of parentheses is considered as one equivalent functionality.

```
es ::= eqvFunc | es - es | es * es | ( es )
```

Fig. 2: EBNF Definition for Execution Strategy (es)

Here are some examples to help understand the formulation:

- Given two equivalent functions a and b, $a - b$ expresses that the functions are to be executed in sequence from left to right, while $a * b$ expresses that the functions are to be executed in parallel. If any `eqvFunc` succeeds and returns, the overall execution terminates without requiring to execute the `eqvFuncs` later in a strategy expression (i.e., for $a - b$, if a succeeds, no need to execute b).
- The parentheses operators denote that the execution plan inside a pair of parentheses is considered as one equivalent functionality. For example, $a*b-c$ means to execute a and b in parallel first and then c, while $a*(b-c)$ means to treat $b - c$ as an equivalent functionality, and execute a and $b - c$ in parallel.
- Notice that because the $-$ and $*$ operators take equivalent functionalities as their operands, the traditional built-in operator precedence is slightly altered. For example, for

the execution plan $a - b * c$, a is executed first; then b and c are executed in parallel.

According to our formulation, the four execution strategies given in the front of this section can be expressed as:

```
1 a - b - c - d - e; //fail-over
2 a * b * c * d * e; //speculative parallel
3 a * b - c * d * e; // or: b * a - c * e * d
4 a - b * c - d - e; // or: a - (b*c) - d - e
```

Fig. 3: Execution Strategy Examples

We notice that in some cases, the parentheses in some cases can be removed (Line 4, Fig. 3) or the sequence of `eqvFuncs` can be changed (Line 3, Fig. 3), without affecting the execution semantics of a strategy. Here we give three observed properties for an execution strategy, which will be used to remove replications in determining all possible execution strategies in the next subsection:

Observation 1. *The parallel operation is commutative, while the sequential one is not, e.g.:* $a*b = b*a$, *while* $a-b \neq b-a$.

Whether two execution plan expressions are equivalent depends on whether they express the same execution control logic. $a*b$ means to execute a and b in parallel, while $b*a$ also means to execute a and b in parallel. In contrast, $a-b$ means to execute a first and then b, while $b-a$ means to execute b first and then a. Hence, $a*b = b*a$, while $a-b \neq b-a$.

Observation 2. *Both the parallel and sequential operators are associative, e.g.:* $a-b-c = (a-b)-c = a-(b-c)$, *and* $a*b*c = (a*b)*c = a*(b*c)$.

$a-b-c$ means to execute a first, then b, and then c. $(a-b)-c$ and $a-(b-c)$ express exactly the same execution control logic. The same argument applies to $a*b*c$, $(a*b)*c$ and $a*(b*c)$.

Observation 3. *Parenthesis are only required to disambiguate expressions that contain the "$-$" operator (not nested in other parenthesis), with the "$*$" operator appearing right before or after the expression's parenthesis. E.g.:* $(a-b)*c \neq a-b*c$, $(a*b-c)*d \neq a*b-c*d$, *while* $a-(b*c) = a-b*c$.

There are three different possible operator combinations inside and outside parenthesized expressions: (1) no un-nested $-$ inside, e.g., $(a*b)-c$, $a-(b*c)-d$, $(a*b)*c$, $a*(b*c)*d$, $a-(b*c)*d$, or $\big((a-b)*c\big)-d$ (the outside parenthesis); (2) an un-nested $-$ inside, with no direct connections to $*$ right outside, e.g., $(a*b-c)-d$, or $a-(b-c)-d$; (3) an un-nested $-$ inside, with at least one connected $*$ right outside, e.g., $(a-b)*c$, $a*(b-c)-d$, or $a-(b-c)*d$. The parentheses in (1) and (2) can be removed based on observations 1 and 2 above. However, the parentheses in (3) cannot be removed, as $(a-b)*c$ and $a-b*c$ expresses different execution logic: $(a-b)*c$ executes a and c in parallel, and will not execute b unless a returns a failure before c successfully returns; $a-b*c$ executes a first and then b and c in parallel.

B. Determining all Possible Strategies

The problem we solve in this subsection is, *given a number of equivalent microservices (i.e., 3 microservices a, b, and c), how to find all possible strategies to execute them?*

Our solution is inspired by the exhaustive search solution for the 24 game, which is a classic math game: given 4 numbers in the range from 1 to 9, binary operators (+, -, *, /), and parentheses, form an arithmetic expression that equals to 24. The exhaustive search solution [35] lists all possible expressions and removes duplicates. To generate all expressions, the solution proceeds in three steps: 1) put all digits into 4 slots, resulting in P(9, 4) arrangements; 2) for each arrangement, put any one of the 4 operators into each of the 3 slots between the digits, resulting in (P(9, 4)*4^3) arrangements; 3) to process parentheses, alter the precedence of the 3 operator slots. The number of final expressions is P(9, 4)*4^3*P(3,3).

We convert the problem of "finding all possible execution plans for an equivalent set of size n" to "finding all execution plan equations that contain m ($1 \leq m \leq n$) equivalent functionalities out of n, with $m-1$ operators out of $-$ and $*$, and parentheses." We first apply the aforementioned exhaustive search to find $P(n,m)*2^{m-1}*(m-1)!$ expressions of execution plans for all m $\in [1, n]$, remove the duplicate expressions, and put them together to produce the answer. The duplication removal procedure takes advantage of the three observations above to identify duplicate expressions.

For each $2 \leq n \leq 6$, Table I gives the number of distinct execution strategies for an equivalent set of size n. $F(M)$ denotes the size of strategies that contains all M microservices, while $F'(M)$ denotes the size of the strategies that contains 1 to $|M|$ microservices. We observe that as few as four equivalent functions can have over 200 possible execution strategies.

M	2	3	4	5	6
$F(M)$ (with M microservices)	3	19	207	3211	64743
$F'(M)$ ($\forall n \in [1, M]$ ms)	5	31	305	4471	87545

TABLE I: Execution Strategies for M Eqv MS

C. Estimating the QoS of a Strategy

This subsection presents a solution to the following problem: *given the QoS of a, b, c, what is the average QoS of executing $a*b*c$ multiple times?* Some existing solutions estimate the overall QoS of an execution strategy by folding the QoS calculation over a collection of equivalent services [15]. We will compare our results with theirs. We are not estimating the QoS of one execution, as any microservice could fail or succeed, leading to dissimilar performance. Instead, we estimate the **average** QoS of running an execution strategy multiple times.

1) QoS Model and Assumptions: We consider three major QoS attributes for edge services and microservices: **cost**, **latency** (or say, response time), and **reliability**. The cost attribute is estimated as the amount of energy consumed to execute edge microservices. The latency attribute refers to the time taken to execute microservices and services. The reliability attribute refers to the probability of finishing an execution successfully. As the execution status in edge environments differs across runs, we compute the QoS as the average value of multiple executions in an edge environment.

$\mathcal{M} = \{m = 1, 2, ...M\}$ denotes a set of equivalent microservices, while r_m, l_m, c_m denote the average reliability, latency, and cost of $\forall m \in \mathcal{M}$ in an edge environment. Our QoS estimation is based on the following assumptions:

Assumption 1: although multiple devices can provide a microservice in an edge environment, our system only selects the one with the best QoS;

Assumption 2: once an edge device receives a microservice execution request, it charges the microservice's full execution cost, irrespective of whether the execution is to succeed, fail, or terminate midway.

2) QoS Estimation Algorithm: We estimate the QoS of a strategy as follows. The overall reliability of a strategy can be directly estimated as $r = 1 - \prod_{m}^{m\in\mathcal{M}}(1 - r_m)$, as a strategy only fails when all its constituent microservices fail. For cost and latency, we first convert the expression of an execution strategy to a tree structure, which has three node types: `leaf`, `sequential`, and `parallel`. A `leaf` is an equivalent microservice. A `sequential` node has its `left` and `right` children, and a `parallel` node has two or more `child` nodes.

Algorithm 1 shows how to estimate the cost and latency using a tree. Starting from the `root` of the tree, it recursively calculates the timelines for all microservices (Lines 15 to 33). A timeline ($\tau = (m, s, e)$) denotes a microservice m, its start time s and end time e. For a `leaf` node, m points to its microservice, with start time set to 0 ($s = 0$) and end time set to the latency of the microservice ($e = l_m$). For a `sequential` node, the time lines of its `left` and `right` children are generated. The longest end time of microservices belonging to the `left` child is added to the start time and end time of each microservice belonging to the `right` child (Lines 23 to 25), as the right child of a sequential node is only executed when all microservices in the `left` child fail. For a `parallel` node, the timelines of all its children are generated.

Lines 3 to 7 calculate the latency of a strategy. The timelines are sorted by their end time in ascending order to form a list ϕ. The overall latency is calculated as follows: for each microservice ($\phi(i)$), add up its end time multiplied by the probability that the overall execution terminates upon the microservice completing its execution (the probability that all microservices in front of $\phi(i)$ fail and $\phi(i)$ succeeds). Lines 9 to 12 calculate the cost of a strategy. The overall cost is calculated as follows: for each microservice m, add up its c_m multiplied by the probability that the overall execution would not terminate before it has a chance to execute (i.e., all microservices in ξ fail, with ξ denoting all microservices that finish before m starts).

3) QoS Estimation Example: For example, consider the $a * b * c$ strategy, in which $l_a = 10ms$, $r_a = 10\%$, $l_b = 90ms$, $r_b = 90\%$, $l_c = 70ms$, and $r_c = 70\%$. By using our QoS estimation method, the latency of the aforementioned $a * b * c$ would be estimated as: $10 * 10\% + 70 * (1 - 10\%) * 70\% + 90 * (1 - 10\%)(1 - 70\%) = 69.4ms$

The folding based method [15] has also been applied to estimate the QoS of a strategy. It first calculates the latency and reliability attributes of $\theta = a * b$ as: $l_\theta = 10 * 10\% + 90 * (1-10\%) = 82ms$, $r_\theta = 1-(1-10\%)*(1-90\%)$=91%. Then it computes $\theta * c$, leading to an estimated overall latency of $70 * 70\% + 82 * (1-70\%) = 73.6ms$. However, this estimation fails to consider how the execution status of services that appear later on the list affect the execution of services preceding them. If, for example, c successfully completes its execution first, its result will be used right away, without waiting for b to complete its execution. Our evaluation in Section V confirms the correctness of our method.

Algorithm 1 Estimate Cost, Latency for a Strategy

```
Input: $es$: strategy
Output: $l$: latency, $c$: cost
1:  $l \leftarrow 0, c \leftarrow 0$
2:  $\tau \leftarrow$ GetTimelines($es$.root)
3:  $\phi \leftarrow \tau$.sortBy(e)                         ▷ sort by endTime
4:  for $i \leftarrow 0$ to $|\phi| - 2$ do
5:      $l+= \left(\prod_{j=0}^{i}(1 - r_{\phi(j).m})\right) * r_{\phi(i).m} * \phi(i).e$
6:  end for
7:  $l+= \left(\prod_{i=0}^{|\phi|-2}(1 - r_{\phi(i).m})\right) * \phi(|\phi| - 1).e$
8:
9:  for $(m, s, e) \in \tau$ do
10:     $\xi \leftarrow \tau$.filter(_.$e < s$) ▷ any ms finishs before $m$ starts
11:     $c+= \prod_{j=0}^{|\xi|-1}(1 - r_{\xi(j).m}) * c_m$
12: end for
13: return $l, c$
14:
15: function GetTimelines($t$:Tree)($\{\tau = (m, s, e)\}$)
16:     switch $t$.Type do
17:         case Leaf
18:             return $\{(t$.func, 0, $t$.func.latency$)\}$
19:         case SequentialNode
20:             $\tau_l \leftarrow$ GetTimelines($t$.left)
21:             $t_{left} \leftarrow$ max($\tau_l.e$)
22:             $\tau_r \leftarrow$ GetTimelines($t$.right)
23:             for $i \in \tau_r$ do
24:                 $i.e \leftarrow i.e + t_{left}$, $i.s \leftarrow i.s + t_{left}$
25:             end for
26:             return $\tau_l \cup \tau_r$
27:         case ParallelNode
28:             $\tau \leftarrow \emptyset$
29:             for $i \in t$.children do
30:                 $\tau \leftarrow \tau \cup$ GetTimelines($i$)
31:             end for
32:             return $\tau$
33: end function
```

D. Execution Strategy Examples

For the aforementioned fire detection example, we set the QoS, $[cost, latency, reliability]$ of microservices $a - e$ to $[50, 50, 60\%]$, $[100, 100, 60\%]$, $[150, 150, 70\%]$, $[200, 200, 70\%]$, and $[250, 250, 80\%]$. Table II lists example strategies and their resulting QoS, calculated by the methodology introduced in this section. We observe that compared with the predefined strategies (strategies 1 & 2), the customized strategies (strategies 3 & 4) strike better balance between the QoS attribute values. For example, if latency is the major concern, strategy 2 is the most latency-efficient but cost-inefficient, while strategy 4 reduces the cost by 50.6% with a minor increase on the latency (5%). This example demonstrates how executing equivalent microservices by different strategies leads to vastly dissimilar QoS.

id	Execution Strategy	cost	latency	reliability
1	a-b-c-d-e	126	126	99.7%
2	a*b*c*d*e	750	81	99.7%
3	a-b*c-d-e	162	111	99.7%
4	c*(a*b-d*e)	372	85	99.7%

TABLE II: Execution Strategies and Estimated QoS

IV. System Design and Strategy Generation

The design of our edge-based service provisioning system follows and extends that of MOLE [31]. In particular, we extend the edge gateway to support new workflows to provision QoS-consistent edge services.

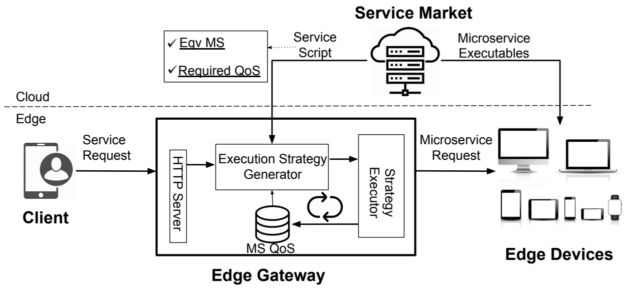


Fig. 4: System Design for Provisioning Edge Services

A. System Components and Edge Service Execution

Fig 4 shows the main components and service provisioning workflow of our design, which features a client, an edge gateway, multiple edge devices for executing microservices, and a cloud-based market that hosts self-describing scripts for services and microservice executables.

A client device sends edge service requests, identified by a unique `ServiceID`, to its connected gateway. The gateway follows a service script describing the dataflow of constituent microservices and the QoS requirements to invoke microservices that are further being executed on edge devices. The service scripts required by the gateway and microservice executables required by the edge devices can be downloaded from a service market, and cached locally for further executions. Hence, if a recently executed service is invoked again, the request can be processed entirely within the edge's local environment, without needing to interact with the cloud.

The runtime starts executing a service by following the default execution strategy to collect the environment-specific non-functional performance attributes for each invoked microservice. As the service continues being invoked, a generator (on the gateway) synthesizes an execution strategy that satisfies the QoS requirements more closely by adapting to the changed performance of the constituent microservices. That strategy executes until a successor with better QoS replaces it, so the system self-adapts to dissimilar edge environments.

B. Major Enhancements Over MOLE

In MOLE, a service script specifies a prioritized list of equivalent microservices. A script is then uploaded to a cloud-based service market to be transformed into an execution strategy, based on the priorities of the constituent equivalent microservices and the developer specified execution strategies. Different from MOLE, our system generates the execution strategies locally at the edge gateway, to accommodate the edge-specific performance of the microservices.

In addition, the edge gateway now involves a feedback loop that comprises an execution strategy generator, a collector for recording microservice QoS characteristics, and a strategy executor. Upon receiving a service request, the gateway imports the corresponding service script, reading the QoS of microservices and the service's QoS requirements. An execution strategy generator retrieves the QoS of constituent microservices from the collector, and outputs an execution strategy. The strategy executor follows the strategy to invoke microservices. The collector keeps updating the QoS characteristics of microservices until their executions complete.

C. QoS Utility Index

The QoS satisfaction model for cloud services is binary: given a set of QoS requirements and a service's SLA, the service either satisfies the requirements or not. Application developers select to integrate only those services that satisfy the QoS requirements. However, with edge applications, developers may have no alternatives and can only use the available edge services, rendering the binary QoS satisfaction model inapplicable. Although QoS requirements are still imposed, applications may need to integrate with edge services that approximate the requirements most closely, and that is what our strategy generation aims for. If a generated strategy fails to reach one or multiple required QoS attributes specified in service scripts, the gateway reports the estimated unsatisfied QoS to the client, which then determines whether the service request with this expected QoS should be continued.

QoS has multiple attributes. For generality, we consider $\mathcal{N}$ QoS attributes, with $n = |\mathcal{N}|$. For example, in our system model, $\mathcal{N} = \{c, l, r\}$, so $n = 3$. $\mathcal{Q}_n$ denotes the requirement of QoS attribute n imposed on an edge service. In our system model, Q_r, Q_c, Q_l denote the requirements on reliability, cost, and latency, respectively. $S = \{s = 1, 2, ..., |F(M)|\}$ denotes all possible strategies, while $Q(s) = \{q_1(s), q_2(s), ..., q_n(s)\}$

denotes the estimated QoS of strategy s. QoS attributes can be placed in the following two categories, as based on their optimization criteria: 1) the smaller the better, denoted as $\mathcal{N}_-$ (i.e., cost and latency); 2) the higher the better, denoted as $\mathcal{N}_+$ (i.e., reliability and trust level). For any QoS attribute $n \in \mathcal{N}$, $q_n \preceq q'_n$ denotes q_n is worse than or equals to q'_n (i.e., $q_n \leq q'_n$ if $n \in \mathcal{N}_+$, or $q'_n \leq q_n$ if $n \in \mathcal{N}_-$), and $q_n \succ q'_n$ denotes q_n is better than q'_n.

Among S, the QoS of a subset of strategies are Pareto optimal [25]. A strategy s is Pareto optimal `iff` no other strategies in $\mathcal{S}$ can improve any of the QoS attributes without worsening the remaining QoS attributes $\big($i.e., $\nexists s' \in \mathcal{S}$, that satisfies: $\forall n \in \mathcal{N}, q_n(s) \preceq q_n(s')$ and $\exists n \in \mathcal{N}, q_n(s') \succ q_n(s)\big)$. To evaluate how these Pareto optimal strategies satisfy the QoS requirements, we introduce a utility index $U(s) = \sum_n^{\mathcal{N}} u_n(s)$, where

$$u_n(s) = \begin{cases} -k\dfrac{|q_n(s) - Q_n|}{Q_n}, \text{if } q_n(s) \preceq Q_n \\ \dfrac{|q_n(s) - Q_n|}{Q_n}, \text{if } q_n(s) \succ Q_n \end{cases} \forall n \in \mathcal{N}, k > 1 \tag{1}$$

In the equation above, $\frac{|q_n(s) - Q_n|}{Q_n}$ denotes the normalized distance between a strategy's estimated value and the requirement imposed on the QoS attribute n. $u_n(s)$ is positive when $q_n(s) \succ Q_n$, negative when $q_n(s) \prec Q_n$, and zero otherwise. However, when the requirement is not satisfied (i.e., $q_n(s) \succ Q_n$), $u_n(s)$ changes at a higher rate due to the system parameter k. The reasoning behind this index is that even for a fully satisfied QoS attribute, its improvement can still increase the overall utility; however, the rate of the increase would be slower than when the QoS attribute is unsatisfied.

To demonstrate how the utility index metric works, consider two strategies s_1 and s_2. s_1 delivers exactly the required cost, latency, and reliability, while s_2 improves cost and reliability by 5% each at the expense of 10% additional latency. With k as a penalty for unsatisfied QoS attributes, the utility of s_1 is higher than that of s_2. A higher k value can be specified to incur a higher penalty for unsatisfied QoS attributes. For example, assume s_2 improves cost and reliability by 10% each at the expense of 10% additional latency; hence, $u(s_1) = u(s_2) = 0$ if $k = 2$, while $u(s_1) = 0 > u(s_2) = -0.1$ if $k = 3$.

D. Generation Heuristic

The pseudo code in Alg. 2 shows our strategy generation heuristic. To generate execution strategies time-efficiently, we use the exhaustive search when the number of equivalent microservices is small, and switch to an approximation heuristic when the number exceeds a threshold causing the exhaustive search to take too long to finish. For a set of M equivalent microservices, the exhaustive search estimates the QoS performance for each possible execution strategy that contains all M microservices (i.e., $F(M)$), and selects the one with the highest utility index (i.e., $\arg\max U(s), \forall s \in F(M)$). However, as the number of possible execution strategies grows exponentially with the number of equivalent microservices, estimating the QoS for each of them may take too long.

Algorithm 2 Strategy Generation

```
Input: $\mathcal{M}$: equivalent microservices
Output: $es$: execution strategy
1:  if $|\mathcal{M}| > \theta$ then
2:      $es \leftarrow$ exhaustiveSearch(strategies($|\mathcal{M}|$))
3:  else
4:      $\mathcal{M}' \leftarrow sortByUtility(\mathcal{M})$
5:      $es \leftarrow \mathcal{M}'(0)$
6:      for $i \leftarrow 1$ to $|\mathcal{M}'| - 1$ do
7:          $es_1 \leftarrow es - \mathcal{M}'(i)$ , $es_2 \leftarrow (es) * \mathcal{M}'(i)$
8:          if utility($es_1$)>utility($es_2$) then
9:              $es \leftarrow es_1$
10:         else
11:             $es \leftarrow es_2$
12:         end if
13:     end for
14: end if
15: return $es$
```

The approximation heuristic first sorts the equivalent microservices by their utility values (i.e., the microservices appear in the order of their overall performance). The initial execution strategy only includes the first microservice from that list. Then, in each iteration, the first microservice on the list is removed and included into the strategy, thus passing through the entire list.

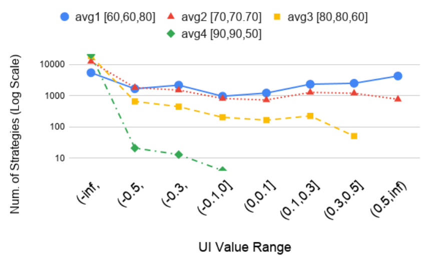
Both algorithms generate strategies that contain all M equivalent microservices. Another generation heuristics could generate strategies that contain only a subset of these microservices. The exhaustive search can include all possible strategies $F'(M)$ (with 1 to M microservices) instead of $F(M)$, while the approximation heuristic can terminate when including a microservice into a strategy fails to improve the utility index. However, as the execution resources in an environment may change over time, executing a generated plan that includes only a subset of equivalent microservices may cause the remaining microservices to stay excluded from being executed. If an originally included microservice becomes unavailable, the strategy generator may fail to switch to an alternative superior strategy, due to the lack of historical execution data for the microservices excluded from the original strategy.

V. Reference Implementation and Evaluation

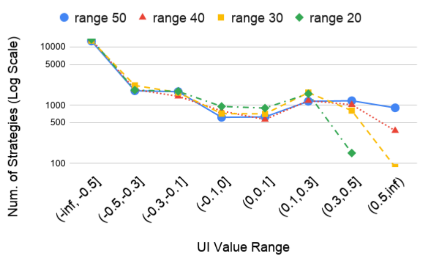
Our evaluation seeks answers to the following questions:

- Does changing execution strategies substantially impact QoS?
- Is our QoS estimation accurate? How does our generated strategy compare with the predefined strategies in terms of their estimated QoS?
- How does the approximation heuristic perform compared with the exhaustive search?
- How does our system perform in real setups? Does it outperform MOLE in dissimilar edge environments?

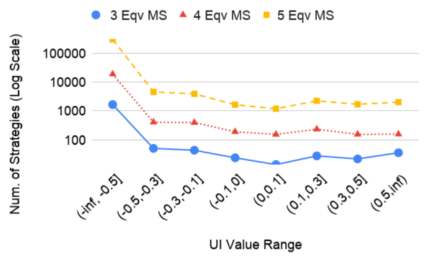
In the following, our evaluation confirms that our QoS estimation can reliably predict the expected service perfor-



(a) Exp1: Varying avg [c, l, r]



(b) Exp2: Varying QoS Range(Δ)



(c) Exp3: Varying Number of Eqv. Microservices

Fig. 5: Utility Distributions of all Possible Strategies for Exp1, Exp2, and Exp3

Exp ID	Config ID	Num of Eqv MS	avg c, l, r	Δ
exp1	1	4	60, 60, 80	50
	2		70, 70, 70	
	3		80, 80, 60	
	4		90, 90, 50	
exp2	1	4	70, 70, 70	50
	2			40
	3			30
	4			20
exp3	1	3	90, 90, 50	100
	2	4		
	2	5		

TABLE III: Simulation Configurations

mance. Compared with the predefined strategies, our generated strategies increase the ratio of QoS-satisfied services by 2× for fewer than 5 equivalent microservices, and by 2.6 × for 5 to 10 equivalent microservices. In a given edge environment, our system outperforms MOLE in terms of cost, latency, and reliability by 31%, 52%, and 4%, respectively. Besides, our system dynamically optimizes the overall QoS by adapting to the resource changes of edge environments.

A. Simulation

The simulation runs on a ThinkCentre M900 Tiny desktop (i7-6700T CPU and 32G memory). We randomly assign QoS values to a number of equivalent microservices.

1) Utility of all Possible Execution Strategies: As shown in Table III, we conduct three sets of experiments, $exp1$, $exp2$, $exp3$, each with a number of configurations. We use [c, l, r] to denote the average value of cost, latency, and reliability, respectively, and use Δ to denote the value range (e.g., $cost = rand(c - \frac{\Delta}{2}, c + \frac{\Delta}{2})$). For each configuration, we simulate 100 services. The QoS requirements in all three experiments are $Q_c = 100$ (units), $Q_l = 100$ (ms), $Q_r = 97$ (%).

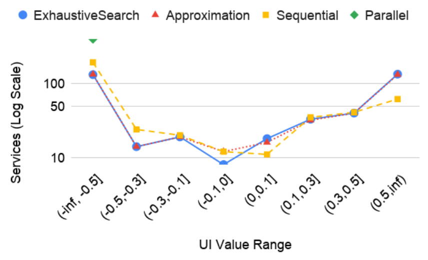
For $exp1$, $exp2$, and $exp3$, Fig. 5.(a, b, c) show the utility distribution of all possible strategies for all randomly generated 100 services in each configuration, respectively. Different lines in each graph denote different configurations. In general, we observe that for all configurations, different execution strategies lead to vastly dissimilar utilities. With higher average QoS, higher Δ (the varying range of QoS), and more equivalent microservices, more execution strategies show higher utility index values.

2) Correctness of QoS Estimation: We randomly select 100 execution strategies from different configurations, and compare their execution performance with our QoS estimations. We use `system.sleep` to imitate each microservice's execution latency, with each strategy executed 300 times. To filter out the costs of scheduling multi-threaded executions, we use "second" as the latency unit of microservices. For example, to verify the execution latency of $a*b*c$ with the QoS settings in Section III.C, we set the average execution time of a, b, and c to 10, 90, and 70 seconds, respectively, and then observe the average overall execution latency of 69.43 seconds. For the other executions, the difference between the average execution latency and our estimations are less than 1%, thus confirming the correctness of our QoS estimation.

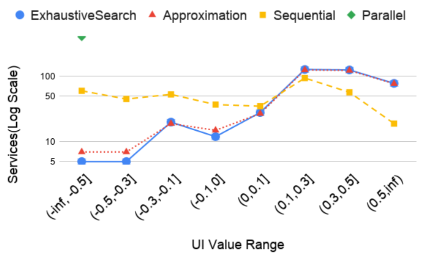
3) Comparing Generated and Predefined Strategies: Then, we calculate the utility values of strategies generated by the exhaustive search and approximation heuristics, and those of the predefined sequential and parallel strategies, as shown in Fig. 6.(a, b, c). From the UI value distribution, we observe that: 1) our strategies obviously outperform the predefined strategies for all three experiments, as more of their utilities fall into the range of high values; 2) the exhaustive search and $Approximation$ produce strategies with comparable performance in terms of their utility values. Fig. 6.(d, e) show the number of services whose QoS requirements are satisfied and the average utility values of various generation heuristics under each configuration. Compared with the predefined strategies, our heuristic increases the ratio of QoS-satisfied services by an average of 2×.

Besides, from Fig.6.d we also observe that the overall performance of the generated strategies is impacted by the number of equivalent microservices and their average performance, but is not impacted by the QoS range (Δ) of these microservices.

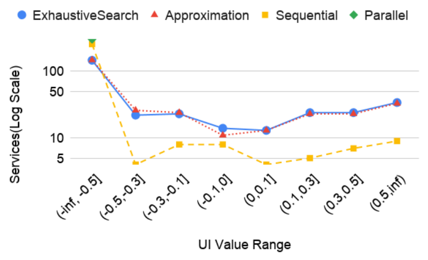
4) Comparing Exhaustive Search and Approximation: To evaluate how our generation heuristic scales, Fig. 7 shows the performance of edge services with more than 5 equivalent microservices. Fig. 7.(a) shows the generation time of different algorithms. With the increase of equivalent microservices in an edge service, the exhaustive search's time increases exponentially, while the time taken by the approximation heuristic and that by the default strategy (either sequential or parallel, represented as a tree) increase only moderately. Fig. 7.(b,c)



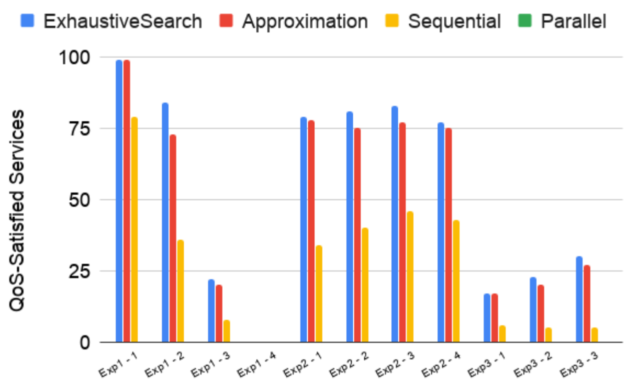
(a) UI Distribution for Exp1



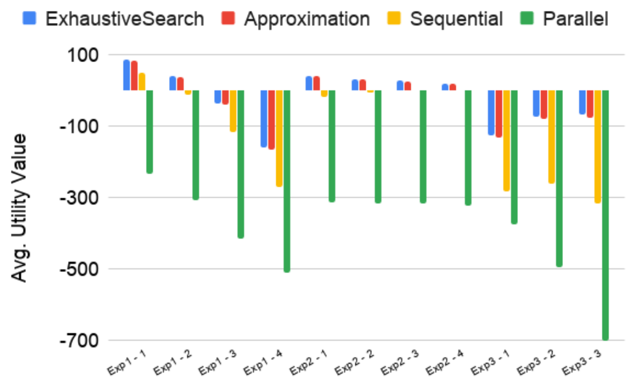
(b) UI Distribution for Exp2



(c) UI Distribution for Exp3

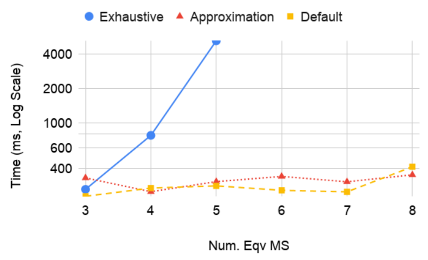


(d) Number of Services with Fully Satisfied QoS of Different Generated Strategies

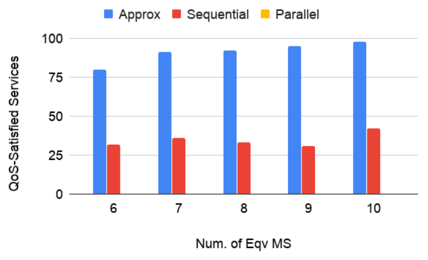


(e) Average Utility Values of Generated Strategies

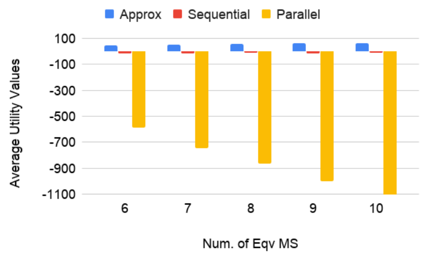
Fig. 6: Utility Distributions of Generated and Predefined Strategies



(a) Strategy Generation Time



(b) QoS Satisfaction Ratio of Strategies for More than 5 Eqv MS



(c) Utility Values of Strategies for More than 5 Eqv MS

Fig. 7: Performance for Generating Strategies for More than 5 Equivalent Microservices

show the number of QoS-satisfied services and the average utility values of different strategies. Hence, as the number of equivalent microservices increases, our generator continues outperforming the predefined strategies (2.6 $\times$ QoS-satisfied services) without incurring much additional execution latency (10% extra time).

B. System Performance

To support the cross-platform deployment on edge gateways, our runtime system is implemented in Java. In our experimental setup, a ThinkCentre M900 Tiny desktop (i7-6700T CPU and 32G memory) serves as the gateway, while a Raspberry Pi 3 (BCM2837 CPU and 1G RAM) and two ThinkCentre M92p Tiny desktops (i5-6500T CPU and 8G memory) serve as edge devices. Each edge device registers its available microservices and their usage costs with the gateway.

To compare with MOLE, we reimplement its evaluation use cases. Three microservices are deployed to detect the ambient temperature, including 1) read a DS1820 temperature sensor; (`readTempSensor`) 2) read a CPU temperature sensor and estimate the environmental temperature [20] (estTemp); 3) query a web service for the location of the current IP address, and query another web service for the location's temperature (readLocTemp). We deploy `readTempSensor` on the Raspberry Pi with a DS1820 sensor connected via GPIO. The execution time for reading the DS1820 sensor is around 950ms, so the microservice reads the sensor every 30 seconds, caches the results, and uses the cached readings as output.

`estTemp` and `readLocTemp` are deployed separately on the two M92p Tiny desktops.

We simulate 100 service invocations per a time slot. In the first time slot, the gateway has no previous microservice execution history, so it follows the **default** speculative parallel strategy. In the next time slots, the gateway uses the execution records in the previous time slot to generate execution strategies and execute them. We set the reliability of these three microservices to 70%, and their cost to 50. The generated strategy is "`readTempSensor-estTemp-readLocTemp`". Table IV shows the execution results. We observe that: 1) the measured QoS of the generated strategy is better than that of the default strategy; 2) the difference between the measured QoS and the estimated QoS is minor.

QoS	Default Strategy	Estimation of Gen. Strategy	Measured
cost	100	70	69
latency	163	81	78
reliability	94	97	98

TABLE IV: Execution Results of Setting 1

We further show how our system adapts to the changes in microservice QoS in an edge environment. We adopt the microservice QoS and service QoS requirements of the setting above, and emulate the resource change by: 1) after being executed 230 times (a randomly selected number), the reliability of `readTempSensor` drops to 20%; 2) after being executed 430 times, the reliability of `readTempSensor` recovers back to 70%. Figure 8 shows the QoS of different time slots, each comprising 100 executions. The execution strategy generated after executing the default speculative parallel strategy is `readTempSensor-estTemp-readLocTemp`. At time slot 1, the reliability of `readTempSensor` drops to 20%. Hence, the execution strategy for slots 2 to 5 is `estTemp-readTempSensor-readLocTemp`. Then, the reliability of `readTempSensor` recovers at slot 5, so the execution strategy for slots 6 and 7 gets back to the previous strategy. We observe that: 1) the QoS of slots 2, 3 and 4 is better than that of slot 1; 2) the QoS of slots 6 and 7 is better than that of slot 5. This experiment shows that switching between the execution strategies of equivalent microservices of an edge service indeed adapts to the QoS fluctuations of these microservices.

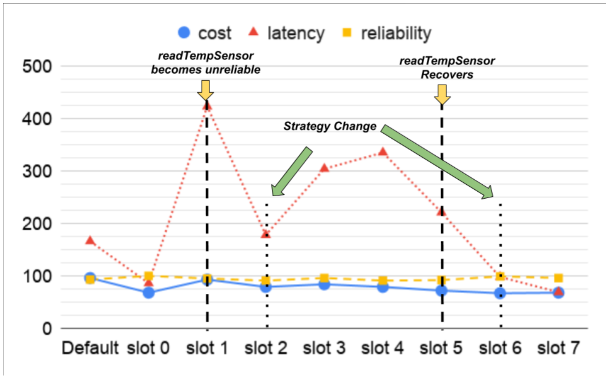


Fig. 8: Average QoS in Different Runs

VI. Related Work

The resources in edge environments are typically scarce, unreliable, and dynamic. To guarantee QoS with scarce resources, most recent edge system designs [24], [29], [33], [38], [39] take advantage of remote resources by offloading computationally intensive tasks to the cloud or nearby edges. To improve reliability and handle unpredictable failures in edge networks, [17] deploys redundant resources as fail-over backups. To adapt to resource dynamicity across edge environments, [16], [18], [34] dynamically adjusts the computational load of edge-based executions by controlling their runtime parameters to fit the available resource budgets, while [22], [37] provide uniform interface to abstract dissimilar hardware and their capabilities.

However, none of the aforementioned designs would be applicable under the following constraints. Remote resources cannot be relied on in the absence of network connectivity or when the local context is required; redundant identical resources may not always be deployed in resource scarce environments; configuring executions self-adaptively may incur runtime failures. To the best our knowledge, our own MOLE [31] is the first attempt to exploit the widely occurring resource/functionality equivalence in edge environments to address the resource scarcity and execution unreliability issues. Instead of relying on identical resources to recover from failures, MOLE relies on resources that provide equivalent functionalities. However, MOLE cannot customize execution strategies on demand to adapt for dissimilar resources across edge environments.

Having not been explored in edge computing, web service compositions apply the combined execution of equivalent services [6], [14], [15], albeit with crude-grained QoS estimation methods. Our work improves the precision of estimating the QoS of execution strategies. To compose equivalent web services, a utility function in [14] normalizes the utility of a QoS attribute by considering its lowest and highest values across all services. In contrast, our utility index normalizes the utility of a QoS attribute in accordance with its QoS requirements, so as to avoid being impacted by the QoS attribute outliers of equivalent microservices.

VII. Conclusion

This paper introduces a novel system design that provides edge services with best effort QoS. Our design improves reliability by executing equivalent functionalities and adapts to resource dissimilarity by varying execution strategies. Through a feedback loop, our design generates environment-specific strategies on demand. As an alternative to adding additional resources, our system design provides best effort edge services by better utilizing the unreliable and dynamic resources at hand. For future work, we plan to apply our system design to improve the scalability and trustworthiness of edge services. Edge systems could invoke equivalent microservices to process multiple concurrent service requests that rely on the same execution resources but are bound by their scarcity, or to protect from malicious devices that return fake results.

Acknowledgment

This research is supported by the National Science Foundation through the Grant CCF-1717065.

References

[1] H. Atlam, R. Walters, and G. Wills. Fog computing and the internet of things: a review. *big data and cognitive computing*, 2(2):10, 2018.

[2] S. Bagchi, M.-B. Siddiqui, P. Wood, and H. Zhang. Dependability in edge computing. *Communications of the ACM*, 63(1):58–66, 2019.

[3] V. Balasubramanian, M. Aloqaily, F. Zaman, and Y. Jararweh. Exploring computing at the edge: a multi-interface system architecture enabled mobile device cloud. In *2018 IEEE 7th International Conference on Cloud Networking (CloudNet)*, pages 1–4. IEEE, 2018.

[4] A. Brogi and S. Forti. QoS-aware deployment of IoT applications through the fog. *IEEE Internet of Things Journal*, 4(5):1185–1192, 2017.

[5] Y. Brun, G. D. M. Serugendo, C. Gacek, H. Giese, H. Kienle, M. Litoiu, H. Müller, M. Pezzè, and M. Shaw. Engineering self-adaptive systems through feedback loops. In *Software engineering for self-adaptive systems*, pages 48–70. Springer, 2009.

[6] V. Cardellini, E. Casalicchio, V. Grassi, S. Iannucci, F. L. Presti, and R. Mirandola. Moses: A framework for qos driven runtime adaptation of service-oriented systems. *IEEE Transactions on Software Engineering*, 38(5):1138–1159, 2011.

[7] F. Carpio, A. Jukan, R. Sosa, and A. J. Ferrer. Engineering a QoS provider mechanism for edge computing with deep reinforcement learning. *arXiv preprint arXiv:1905.00785*, 2019.

[8] A. Carrega, M. Repetto, G. Robino, and G. Portomauro. Openstack extensions for QoS and energy efficiency in edge computing. In *2018 Third International Conference on Fog and Mobile Edge Computing (FMEC)*, pages 50–57. IEEE, 2018.

[9] A. Carzaniga, A. Gorla, N. Perino, and M. Pezze. Automatic workarounds: Exploiting the intrinsic redundancy of web applications. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 24(3):16, 2015.

[10] K. Chen, C. Wang, Z. Yin, H. Jiang, and G. Tan. Slide: Towards fast and accurate mobile fingerprinting for wi-fi indoor positioning systems. *IEEE Sensors Journal*, 18(3):1213–1223, 2017.

[11] N. K. Giang, R. Lea, M. Blackstock, and V. C. Leung. Fog at the edge: Experiences building an edge computing platform. In *2018 IEEE International Conference on Edge Computing (EDGE)*, pages 9–16. IEEE, 2018.

[12] M. Gorlatova, J. Sarik, G. Grebla, M. Cong, I. Kymissis, and G. Zussman. Movers and shakers: Kinetic energy harvesting for the internet of things. *IEEE Journal on Selected Areas in Communications*, 33(8):1624–1639, 2015.

[13] H. Guo, J. Huai, H. Li, T. Deng, Y. Li, and Z. Du. Angel: Optimal configuration for high available service composition. In *IEEE International Conference on Web Services (ICWS 2007)*, pages 280–287. IEEE, 2007.

[14] Y. Guo, S. Wang, K.-S. Wong, and M. H. Kim. Skyline service selection approach based on qos prediction. *International Journal of Web and Grid Services*, 13(4):425–447, 2017.

[15] N. Hiratsuka, F. Ishikawa, and S. Honiden. Service selection with combinational use of functionally-equivalent services. In *Web Services (ICWS), 2011 IEEE International Conference on*, pages 97–104. IEEE, 2011.

[16] M. Hu, L. Zhuang, D. Wu, Y. Zhou, X. Chen, and L. Xiao. Learning driven computation offloading for asymmetrically informed edge computing. *IEEE Transactions on Parallel and Distributed Systems*, 2019.

[17] H. Huang and S. Guo. Proactive failure recovery for nfv in distributed edge computing. *IEEE Communications Magazine*, 57(5):131–137, 2019.

[18] A. Jonathan, A. Chandra, and J. Weissman. Locality-aware load sharing in mobile cloud computing. In *Proceedings of the10th International Conference on Utility and Cloud Computing*, pages 141–150. ACM, 2017.

[19] T. Kauffmann. Solar power for raspberry pi. https://blog.voltaicsystems.com/powering-a-raspberry-pi-from-solar-power/, 2017.

[20] C. Krintz, R. Wolski, N. Golubovic, and F. Bakir. Estimating outdoor temperature from cpu temperature for iot applications in agriculture. In *International Conference on the Internet of Things*, 2018.

[21] M. Le, Z. Song, Y.-W. Kwon, and E. Tilevich. Reliable and efficient mobile edge computing in highly dynamic and volatile environments. In *2017 Second International Conference on Fog and Mobile Edge Computing (FMEC)*, pages 113–120. IEEE, 2017.

[22] Y. Li and W. Gao. Interconnecting heterogeneous devices in the personal mobile cloud. In *IEEE INFOCOM 2017-IEEE Conference on Computer Communications*, pages 1–9. IEEE, 2017.

[23] B. Lucia, V. Balaji, A. Colin, K. Maeng, and E. Ruppel. Intermittent computing: Challenges and opportunities. In *2nd Summit on Advances in Programming Languages (SNAPL 2017)*. Schloss Dagstuhl-Leibniz-Zentrum fuer Informatik, 2017.

[24] Y. Mao, J. Zhang, and K. B. Letaief. Dynamic computation offloading for mobile-edge computing with energy harvesting devices. *IEEE Journal on Selected Areas in Communications*, 34(12):3590–3605, 2016.

[25] R. T. Marler and J. S. Arora. Survey of multi-objective optimization methods for engineering. *Structural and multidisciplinary optimization*, 26(6):369–395, 2004.

[26] A. Mtibaa, A. Fahim, K. A. Harras, and M. H. Ammar. Towards resource sharing in mobile device clouds: Power balancing across mobile devices. In *ACM SIGCOMM Computer Communication Review*, volume 43, pages 51–56. ACM, 2013.

[27] Z. Qiu. Enhancing response time and reliability via speculative replication and redundancy. *Ph.d Thesis*, 2016.

[28] S. Raponi, M. Caprolu, and R. Di Pietro. Intrusion detection at the network edge: Solutions, limitations, and future directions. In *International Conference on Edge Computing*, pages 59–75. Springer, 2019.

[29] A. Samanta, Z. Chang, and Z. Han. Latency-oblivious distributed task scheduling for mobile edge computing. In *2018 IEEE Global Communications Conference (GLOBECOM)*, pages 1–7. IEEE, 2018.

[30] W. Shi, J. Cao, Q. Zhang, Y. Li, and L. Xu. Edge computing: Vision and challenges. *IEEE Internet of Things Journal*, 3(5):637–646, 2016.

[31] Z. Song and E. Tilevich. A programming model for reliable and efficient edge-based execution under resource variability. In *2019 IEEE International Conference on Edge Computing (EDGE)*, pages 64–71. IEEE, 2019.

[32] W.-T. Tsai, X. Sun, and J. Balasooriya. Service-oriented cloud computing architecture. In *2010 seventh international conference on information technology: new generations*, pages 684–689. IEEE, 2010.

[33] L. Wang, L. Jiao, T. He, J. Li, and M. Mühlhäuser. Service entity placement for social virtual reality applications in edge computing. In *IEEE INFOCOM 2018-IEEE Conference on Computer Communications*, pages 468–476. IEEE, 2018.

[34] S. Wang, T. Tuor, T. Salonidis, K. K. Leung, C. Makaya, T. He, and K. Chan. Adaptive federated learning in resource constrained edge computing systems. *IEEE Journal on Selected Areas in Communications*, 37(6):1205–1221, 2019.

[35] Y. Wang. 24 game solution project on github, 2019. https://github.com/MaigoAkisame/enumerate-expressions.

[36] J. Xu, L. Chen, and S. Ren. Online learning for offloading and autoscaling in energy harvesting mobile edge computing. *IEEE Transactions on Cognitive Communications and Networking*, 3(3):361–373, 2017.

[37] D. Y. Zhang, T. Rashid, X. Li, N. Vance, and D. Wang. Heteroedge: Taming the heterogeneity of edge computing system in social sensing. In *Proceedings of the International Conference on Internet of Things Design and Implementation*, pages 37–48. ACM, 2019.

[38] T. Zhao, S. Zhou, X. Guo, and Z. Niu. Tasks scheduling and resource allocation in heterogeneous cloud for delay-bounded mobile edge computing. In *2017 IEEE International Conference on Communications (ICC)*, pages 1–7. IEEE, 2017.

[39] C. Zhu, J. Tao, G. Pastor, Y. Xiao, Y. Ji, Q. Zhou, Y. Li, and A. Ylä-Jääski. Folo: Latency and quality optimized task allocation in vehicular fog computing. *IEEE Internet of Things Journal*, 2018.