Why Cloud Applications Are not Ready for the Edge (yet)

Chanh Nguyen
Umeå University
Umeå, Sweden
chanh@cs.umu.se

Amardeep Mehta
Umeå University
Umeå, Sweden
amardeep@cs.umu.se

Cristian Klein
Umeå University
Umeå, Sweden
cklein@cs.umu.se

Erik Elmroth
Umeå University
Umeå, Sweden
elmroth@cs.umu.se

ABSTRACT

Mobile Edge Clouds (MECs) are distributed platforms in which distant data-centers are complemented with computing and storage capacity located at the edge of the network. Their wide resource distribution enables MECs to fulfill the need of low latency and high bandwidth to offer an improved user experience.

As modern cloud applications are increasingly architected as collections of small, independently deployable services, they can be flexibly deployed in various configurations that combines resources from both centralized datacenters and edge locations. In principle, such applications should therefore be well-placed to exploit the advantages of MECs so as to reduce service response times.

In this paper, we quantify the benefits of deploying such cloud micro-service applications on MECs. Using two popular benchmarks, we show that, against conventional wisdom, end-to-end latency does not improve significantly even when most application services are deployed in the edge location. We developed a profiler to better understand this phenomenon, allowing us to develop recommendations for adapting applications to MECs. Further, by quantifying the gains of those recommendations, we show that the performance of an application can be made to reach the ideal scenario, in which the latency between an edge datacenter and a remote datacenter has no impact on the application performance.

This work thus presents ways of adapting cloud-native applications to take advantage of MECs and provides guidance for developing MEC-native applications. We believe that both these elements are necessary to drive MEC adoption.

CCS CONCEPTS

• **Networks → Network measurement**; • **Computer systems organization → Cloud computing**; • **Software and its engineering → Software design engineering**.

KEYWORDS

Mobile Edge Clouds, Edge Latency, Mobile Application Development, Micro-service, Profiling

ACM Reference Format:
Chanh Nguyen, Amardeep Mehta, Cristian Klein, and Erik Elmroth. 2019. Why Cloud Applications Are not Ready for the Edge (yet). In *SEC '19: ACM/IEEE Symposium on Edge Computing, November 7–9, 2019, Arlington, VA, USA.* ACM, New York, NY, USA, 14 pages.

SEC '19, November 7–9, 2019, Arlington, VA, USA

ACM ISBN 978-1-4503-6733-2/19/11…$15.00
https://doi.org/10.1145/3318216.3363298

1 INTRODUCTION

Mobile Edge Cloud (MEC) is an emerging paradigm in which data-centers are combined with IT services and cloud computing capabilities at the edge of the network, at close proximity to the end-users such as base stations or WiFi access points [1, 2]. This highly distributed approach allows MEC to provide computation and storage capabilities with much lower latencies than would be achievable with a centralized cloud alone. In addition, MECs also help to minimize the central cloud's ingress bandwidth requirements, reducing the impact of network jitter on overall application latency, and alleviating data privacy concerns.

Cloud application users are very sensitive to a service's latency and tend to prioritize it when selecting cloud services. The longer it takes for a service to respond to a user request, the greater the risk that a customer switches to a competing service. A report by Amazon [3] showed that a 100 ms increase in latency costs them 1% in reduced sales, while a similar study by Akamai [4] concluded that on average, a 1 second delay in page response reduces conversions by 7%, page views by 11%, and customer satisfaction by 16%. Given the importance of latency for the success of cloud applications, it is interesting to consider how MECs could be exploited to further reduce their latency.

Modern cloud applications are increasingly architected as collections of micro-services [5]. The micro-service philosophy advocates constructing software applications as collections of small, independently deployable services that communicate with each other via lightweight mechanisms [6]. Micro-services are very agile, and great for accelerating software development, deployment, and operation practices. Consequently, they have seen widespread industrial adoption in recent years, for example within Amazon and Netflix.

An appealing property of micro-service-based cloud application is their flexible deployment: they can be deployed in various configurations, combining resources in both data-centers and edge locations. One might therefore expect them to be well-placed to benefit from the advantage of MECs. With respect to decentralizing and geographically distributing data in proximity to the user, deploying cloud applications over a MEC can be compared to Content Delivery Networks (CDNs) [7, 8], which are successfully used to improve applications' accessibility, availability, and load times through content replication. However, in most cases, CDNs can only accelerate read-intensive applications such as video streaming and mobile content. Conversely, the deployment of micro-service-based cloud applications over a MEC can potentially improve the performance of both read- and write-intensive workloads.

The goal of this paper is to quantify the benefits that MEC can offer to micro-service-based cloud applications. We empirically

measure performance – specifically, end-to-end latency – under different deployment configurations, using resources from both datacenters and edge locations. The aim is to answer two questions: **Can cloud applications leverage the advantages of MEC to reduce latency?** and **What architectures and characteristics should cloud applications possess to make them ready for MECs?** We believe that these questions must be answered to avoid a "dead-lock" scenario in which a lack of applications slows down MEC investment and a lack of MEC deployments discourages building MEC-native applications (Section 2).

We initially present experiments using an emulated MEC showing that, against conventional wisdom, end-to-end latency does not improve significantly even when most services are deployed in an edge location. To explain these findings, we present further studies that show why modern cloud applications do not benefit from MECs, and suggest improvements in cloud application design that allow the benefits of MECs to be realized. We also quantify the gains of those recommendations and show how to basically convert a cloud-native application into a MEC-native application.

The **contribution** of this paper is three-fold:

(1) We present experiments to empirically evaluate the impact on performance of micro-service-based cloud applications in different deployment configurations that combine a remote centralized data-center with edge locations, using various strategies such as master-slave database replication. Two benchmarks are used in the experiments: The CloudSuite Web Serving [9] and Sock Shop [10] (Section 3).
(2) We present our network communication profiling tool and apply it to the two benchmarks to understand why they do not benefit from MEC deployment (Section 4).
(3) We discuss strategies that cloud application developers can use to make their applications ready for deployment over MECs (Section 4).

By an exhaustive empirical study using two popular benchmarks, we show that deployment on an MEC does not significantly improve end-to-end latency even when most application services are deployed at the edge location. We developed a profiler to better understand the causes of these problems, revealing that they originate from the large numbers of transactions between application services when processing end-user requests. The number of transactions multiplied by the network delay between the edge and the remote centralized cloud causes response times to increase dramatically. To overcome this issue, we suggest ways to modify the engineering of cloud-native applications so they can benefit from deployment on MECs. Implementing these architectural changes could bring the performance of cloud micro-service applications on MECs closer to that expected in an ideal scenarios, i.e., in which the latency between the edge location and the remote datacenter has no impact on application performance.

Our paper paves the way to a faster adoption of MECs by showing how non-MEC-native applications, i.e., applications not specifically designed for MECs, can be adapted to take advantage of these emerging infrastructures. In addition, we provide guidelines for developing MEC-native applications. We believe that both these elements are necessary to drive adoption of MECs because the adoption of traditional clouds proceeded in two stages – first legacy applications were moved into the cloud, realizing some of the benefits of cloud computing, then cloud-native applications were developed, realizing the full benefits. Our expectation is that the adoption of MECs will have to follow the same trajectory.

2 WHY SHOULD MECS BENEFIT NON-MEC-NATIVE APPLICATIONS?

In this section we argue that it is important to study the benefits MECs can offer to applications not specifically built for them, which we refer to as non-MEC-native applications.

Let us start by quickly revisiting the history and current status of traditional clouds. The term "cloud" as it is currently used was popularized by Amazon in 2006, and more rigorously defined by the NIST in 2011 [11]. In 2008, Netflix started migrating their existing code to the cloud, being one of the first massive adopters of public clouds. In the process, they found that considerable savings could be made by specifically building their application for the cloud. Around 2014, the term **cloud-native** applications appeared, used to denote applications that are specifically engineered for clouds and take full advantage of their capabilities, in particular elasticity. The concept of applications consisting of large sets of **microservices**, i.e., loosely-coupled fine-grained services, subsequently provided an architecture suitable for cloud-native applications [12].

Despite the great progress in cloud technologies and the development of strategies for building cloud-native applications, many discussions in 2017 still revolve around moving "legacy" applications (let us call them **non-cloud-native** applications) into the cloud. For example, 25% of the respondents in a 2017 survey perceived legacy applications as a barrier to public cloud adaption [13]. In April 2017, the company Docker – the developer of a container platform that is seen as an enabler of cloud-native applications – announced a partnership with IBM to containerize legacy applications [14]. These observations show that cloud adoption is significantly leveraged by non-cloud-native applications.

It is difficult to imagine what the present would look like if clouds offered no benefits to non-cloud-native applications. We believe that in such a hypothetical world, cloud adoption would have been negligible: without applications benefiting from clouds, cloud providers would have been reluctant to invest in infrastructure, and the lack of cloud infrastructure would have made application providers reluctant to develop for the cloud. In essence, clouds would have been a great idea trapped in an inescapable dead-lock.

On this basis, we argue that it is unrealistic to expect MECs to become successful based solely on **MEC-native** applications, i.e., applications engineered specifically for MECs. Therefore, MEC providers and advocates should also focus on the benefits MECs can offer to **non-MEC-native** applications, i.e., applications not specifically engineered for MECs. Of course, the most promising such applications are cloud-native applications, in particular microservice-based applications with high deployment flexibility. Indeed, cloud-native applications allow an operator to decide at runtime what parts of the application should run in data-centers and what parts should run in edge locations, increasing the chances of benefiting from MECs.

Two commonly cited potential benefits of MECs are lower latencies and lower core network bandwidth consumption. In this work we focus on latency because many end-user-facing cloud-native applications need low end-to-end response times: several studies have identified negative correlations between response times and revenues, as discussed in Section 1. Therefore, we aim to answer two **research questions**:

RQ1 How much can cloud-native applications benefit from latency reduction when deployed on MECs?

RQ2 How should cloud-native applications be engineered to maximize these benefits?

We answer these two questions with empirical evidence derived from experiments on two representative microservice-based applications running on an emulated MEC infrastructure.

3 QUANTIFYING THE LATENCY IMPACT OF DEPLOYMENTS

In this section, we present experiments designed to evaluate the latency reductions that cloud-native applications could achieve when deployed on MECs. To this end, we first describe an emulated MEC infrastructure. We then introduce two microservice-based cloud applications that were selected for deployment on the emulated infrastructure in various configurations, using resources from both datacenters and edge locations. Finally, we present the observed end-to-end latencies of these applications under each studied deployment configuration.

3.1 Emulating a MEC Infrastructure

The MEC infrastructure consists of Data Centers (DCs) at the network edge and a remote centralized cloud that promises low latency for mission critical applications and low costs for bandwidth-hungry applications [15]. Networking technology, such as 4G, can achieve 10 ms Round Trip Time (RTT) between the end-users and the edge layer DC [16]. To quantify the RTT between a centralized cloud DC and the end-users, we measured the RTT for lambda Amazon Web Services (AWS) endpoints [17] from our institution in Europe, as shown in Figure 1. These measurements showed that the average latency can vary between 30 ms and 400 ms depending on geographic locations. For example, the lowest achievable latency is 30 ± 10 ms for round trip from our location to the EU-central-1 location (i.e., Frankfurt, Germany). Based on these measurements, the network RTT delay was set to 10 ms between the end-users and the edge layer. In order to compare the application's performance when deployed in a MEC against the best achievable application's performance when deployed in a centralized cloud, the network RTT delay between the end-users and the remote centralized cloud DC was set to around 40 ms, as shown in Figure 2.

The MEC infrastructure was emulated by using NetEm [18], a Linux network traffic control emulation system, to inject these delays between two layers emulating the remote centralized cloud and edge locations. Note that the bandwidth between network layers is set to a high value (1024 Mbps) so as to minimize the impact of bandwidth constraints on the final measurement. To avoid difficulties associated with controlling for network latency, we conducted all experiments on a local machine with an Intel i7-4790 CPU and 32GB RAM.

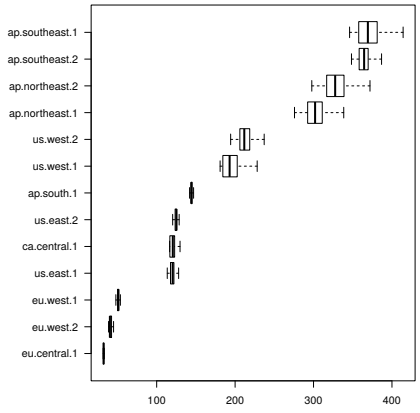


Figure 1: Network round trip time (ms) for selected lambda AWS endpoints from Umeå University, Sweden.

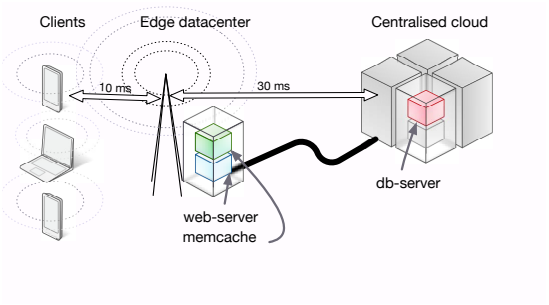


Figure 2: Illustration of an emulated MEC showing the network latencies between clients, an edge datacenter, and the centralized cloud with the Web Serving benchmark deployed in the *Mixed* scenario.

3.2 Application benchmarks

Although microservice architectures are emerging and acquiring acceptance in enterprise IT, most microservice-based cloud applications developed to date are proprietary. Therefore, the research community has relied on plausible microservice application benchmarks to conduct a wide variety of empirical studies on the evaluation and design of such applications. Two such benchmarks are the Cloudsuite [9] Web Serving benchmark and Sock Shop provided by Weaveworks [10], both of which have been widely used. The advantage of these two benchmarks is that container images for all their components are available in Docker repositories [19]. Additionally, their source code is publicly available, making them relatively straightforward to customize and deploy for experimental purposes.

3.2.1 ***Web Serving.*** As shown in Table 1, the Web Serving application benchmark [9] has 4 components: faban-clients, web-server, memcached-server, and db-server. The web-server component is based on the open source social networking engine Elgg, which is used to handle large numbers of highly dynamic requests [20].

Table 1: Web Serving and Sock Shop services

Application	User services	Database services
Web Serving	web-server, memcached-server, faban-client	db-server
Sock Shop	front-end, order, payment, user, catalogue, cart, shipping, queue, queue-Master	orderDB, userDB, catalogueDB, cartDB

Faban-clients are emulated using the Faban framework [21], which generates highly dynamic AJAX requests.

3.2.2 ***Sock Shop.*** Sock Shop is a microservices demonstration benchmark that simulates an online shopping web service for selling socks. As shown in Table 1, Sock Shop has 13 main independent components divided into user-services and database-services. These services are developed using various technologies such as Spring Boot, Go, and Node.js. All Sock Shop services communicate with each other using REST over HTTP [10].

We used the *Docker Compose* tool [22] to deploy Sock Shop and Web Serving, with each independent service being separately deployed in its own Docker container.

3.3 Deployment scenarios

To understand the impact of MECs on the latency of microservice-based applications, we constructed five deployment scenarios for the Web Serving and Sock Shop benchmarks, as shown in Figure 3. The scenarios were:

3.3.1 ***Cloud-Only deployment.*** This is the traditional deployment configuration of microservice-based cloud applications: all application services are hosted in remote large-scale cloud datacenters. In this scenario, the delay between the end-user and all application services is 40 ms.

3.3.2 ***Limited Edge deployment.*** Microservice applications consist of many small independent services. Some of these are user services that communicate intensively with one-another to handle end-user requests. To determine how applications' response times are affected by moving such intensively-communicating user services closer to the end-user, we examined a scenario in which the intensively-communicating user services are deployed at edge locations while all other microservices are deployed at the remote cloud datacenter. The delay between the end-user and the edge-deployed services is 10 ms, while that between the edge-deployed services and those deployed at the remote datacenter is 30 ms. Because the Web Serving benchmark has only one user service (i.e., web-server), we did not test Web Serving in this scenario. Conversely, in the Sock Shop benchmark, there is substantial communication between the front-end service and other user services such as catalogue, user, cart, and order. Therefore, these Sock Shop services are deployed together at the edge location in this scenario.

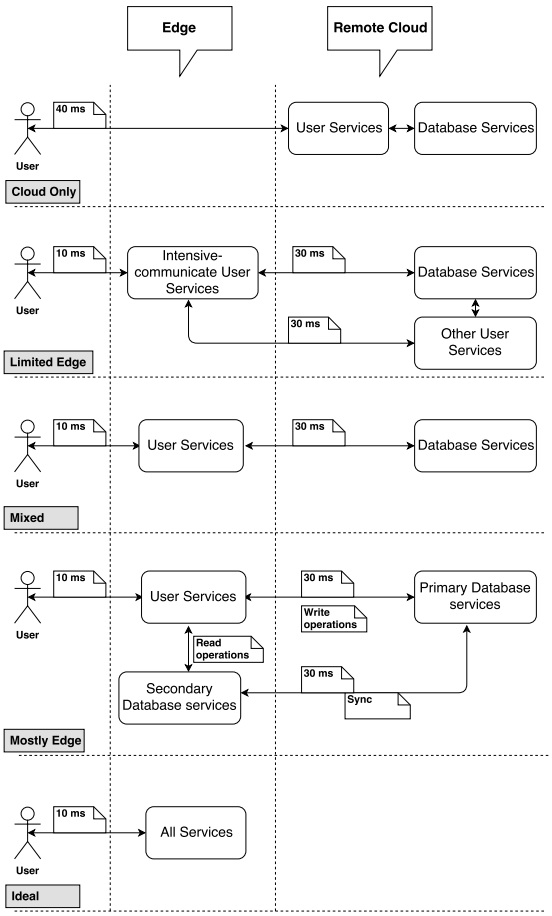


Figure 3: Deployment scenarios used with the Web Serving and Sock Shop benchmarks, showing the network latencies between the application services.

3.3.3 ***Mixed deployment.*** In this scenario, the database services are deployed in remote cloud datacenters, while the user services are hosted at the edge locations. This reduces the network delay between the end-users and user services to 10 ms, while the network delay between user services and database services is 30 ms.

3.3.4 ***Mostly Edge deployment.*** CDN technology has greatly benefited content-delivery-based applications. Put simply, the purpose of a CDN is to to improve applications' accessibility and response times by storing cached versions of their content in multiple geographical locations so as to minimize the distance between the end-user and application server. We constructed a fourth scenario to quantify the response time benefits that read-only workload applications can gain by being deployed on MEC infrastructure using a similar approach. For this purpose, we first customized the design of the Web Serving and Sock Shop applications by creating replicas of their database services. We then classified the requests to the two applications based on the nature of their operations: requests that write new data to the database were classified as write requests, and all others were classified as read requests. The two benchmark applications were then modified such that all read requests were directed

to the secondary databases, while write requests were directed to the primary databases. In this scenario, secondary databases are deployed alongside user services at the edge locations, while the primary databases are hosted on the remote cloud datacenter.

3.3.5 ***Ideal deployment.*** Indeed, in a realistic deployment, the data stored in the edge locations, specifically in the database services, would have to be somehow synchronized, preferably through the remote cloud datacenter, as in the *Mostly Edge* deployment. However, in order to quantify the minimum achievable latency in MECs, we created the *Ideal* scenario: all components of the application are deployed in a single edge location so that the data synchronization latency is completely eliminated. Of course, this scenario is not practicable, but is illustrated here to quantify how far the other deployments are from what we can ideally expect when deploying a cloud application over MECs. With such deployment, the network delay between the end-user and the application is 10 ms.

Sections 3.4 and 3.5 present the measured response times for the Web Serving and Sock Shop applications in the five deployment scenarios outlined above.

3.4 Latency impact on Web Serving

3.4.1 Workload generator. The Faban framework is used to generate workloads for the Elgg social networking engine in the Web Servicing application. It is Java-based and has two main components: Faban driver and Faban harness. Faban driver provides an API that can be used to create different types of requests with specific types of operations and probability matrices. Faban harness deploys and runs the benchmark, and then generates a report containing various statistics such as the mean, standard deviation, and 95th percentile of the response times for the studied request types.

User requests were simulated using the framework in its default configuration. By default, the number of concurrent users is 7, and the time span or duration of a workload is 300 seconds. The requests $R_i, i \in \{1, \ldots, 9\}$ included in the *request mix* are *access home page, login existing user, wall post, send chat message, send friend request, create new user, logout logged in user, update live feed*, and *receive chat message*, as shown in Table 2. The core of Web Serving is the Elgg platform whose operations are AJAX based [20], hence the framework yields many frequent AJAX requests, a large fraction of which keep updating a small part of the web page. Higher probabilities are assigned to more common requests, such as *updating the live feed, posting on walls*, and *sending and receiving chat messages*. Lower probabilities are assigned to less frequent requests, such as *login* and *logout*, *reloading the home page*, and *creating new users*.

3.4.2 Measured response times. Figure 4 shows the results obtained using the Web Serving benchmark application in the four scenarios in which it was tested. The x-axis in this figure represents different request types under all four tested scenarios (since Web Serving has few components, it was not tested under the *Limited Edge* deployment scenario), and the y-axis represents the average response time in seconds.

Table 3 shows the response time increases for the *Mixed*, *Mostly Edge* and *Ideal* deployment scenarios relative to the response times

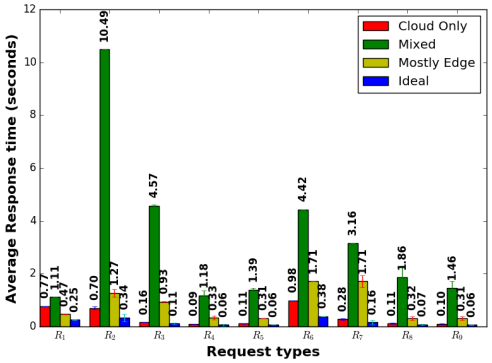


Figure 4: End-to-end response times for Web Serving in four scenarios. Web Serving has only three components, so it was not tested in the *Limited Edge* scenario.

Table 2: The Web Serving request mix and the frequency of each request type in Web Serving workloads [20]. The *Write/All Operations* column describes the percentage of data write operation involved in each request.

SNo	Request Name	Write/ All Operations	Percentage
R_1	access home page	0	5
R_2	login existing user	12	2.5
R_3	wall post	11	20
R_4	send chat message	3	17
R_5	send friend request	3	10
R_6	create new user	12	0.5
R_7	logout logged in user	9.1	2.5
R_8	update live feed	3.3	25.5
R_9	receive chat message	3.3	17

for the *Cloud Only* deployment. These relative increases were calculated using Equation 1, where r_{scenario} is the response time for the scenario in question (i.e., the *Mixed*, *Mostly Edge*, or *Ideal* scenario), and r_{cloud} is the response time for the *Cloud Only* scenario.

$$\frac{r_{\text{scenario}} - r_{\text{cloud}}}{r_{\text{cloud}}} \quad (1)$$

The *Mixed* deployment yielded significant average response time increases of 14×, 27×, 12×, 12×, 10×, 15×, and 14× for the *login existing user, wall post, send chat message, send friend request, logout logged user, update live feed* and *receive chat message* request types, respectively. The *Mostly Edge* deployment yielded smaller relative increases in the average response times for all request types other than *access home page*; the largest relative increases in this scenario were around 5×, and were observed for the *wall post* and *logout logged in user* requests. The explainable for these results is that in these type of requests, there is a percentage of write operations to the primary database which aggregates latency to the overall response time due to the delay between edge and the remote cloud. The *Ideal* deployment yielded lower response times for all requests

Table 3: Relative increases (see Equation 1) in the average response times for different request types in the *Mixed*, *Mostly Edge*, and *Ideal* deployments.

SNo	Request Name	Mixed	Mostly Edge	Ideal
R_1	access home page	0.44	-0.39	-0.68
R_2	login existing user	14.05	0.82	-0.52
R_3	wall post	26.87	4.67	-0.30
R_4	send chat message	11.65	2.52	-0.41
R_5	send friend request	11.65	1.80	-0.43
R_6	create new user	3.52	0.75	-0.61
R_7	logout logged in user	10.39	5.18	-0.43
R_8	update live feed	15.32	1.78	-0.38
R_9	receive chat message	13.56	2.07	-0.41

than the *Cloud Only* deployment because all the components are deployed at the edge rather than the remote centralized cloud.

The *Cloud Only* deployment offered the best performance of the three technically feasible deployment scenarios that were tested (as discussed above, the *Ideal* deployment scenario is not technically feasible). The Web Serving application thus does not benefit from the MEC infrastructure in general; the only case for which benefits were observed was that of type R_1 requests in the *Mostly Edge* deployment scenario.

3.5 Latency impact on Sock Shop

3.5.1 Workload generator. We simulated end-user requests by using the load-test service provided with Sock Shop [23]. Two parameters, i.e., the numbers of requests and concurrent users, must be passed when using load-test to generate requests arriving at Sock Shop.

To generate the *request mix* (i.e., the desired mixture of read and write requests), we used the load-test service's default user behavior configuration. Table 4 presents different request types and the total number of each type generated in a Sock Shop workload with the chosen settings: the number of concurrent users is set to 10, and the total number of requests is set to 1000, respectively.

We customized the user behavior setting to generate the read-only requests. As shown in Table 5, with the chosen parameter settings, the load-test service could be configured to generate 1000 *Get Category* or *Get Basket* requests.

The generated requests were then injected to the Sock Shop application, and its response times were measured under different deployment scenarios.

3.5.2 Measured response times. Figure 5 presents Sock Shop's average end-to-end response times under different deployment scenarios for both the default request mix and read-only requests.

Default request mix. When using the default request mix, the workloads delivered to Sock Shop include both read and write requests. For this mix, the average response time of Sock Shop in the *Cloud Only* deployment scenario was approximately 62 ms. This is understandable because all the application's services are deployed on the remote centralized cloud, for which the network delay to the client is 40 ms. Interestingly, the average response times observed under the *Limited Edge* and *Mixed* deployment scenarios are almost three times those for the *Cloud Only* scenario, even though most user services are deployed at edge locations and are thus closer to the end-user. The *Mostly Edge* deployment yielded a better average response time (approximately 54 ms) than the *Cloud Only* deployment. However, even in this case, the reduction in the response time (8 ms) is smaller than the difference in network latency (30 ms) between the two deployment scenarios. This is presumably due to the presence of write requests that must be sent to database services hosted in the remote centralized cloud data-center.

Notably, the average response time of Sock Shop in the *Ideal* deployment is 27 ms, which is half that for the *Mostly Edge* deployment. While the *Ideal* deployment is impractical, the significant response time difference between the *Ideal* and *Mostly Edge* deployments strongly suggests that the communication between user services and database services is the main factor affecting the increment of the application's response time.

Table 4: Different request types delivered to Sock Shop in the test workload. The total number of requests was set to 1000, including both read and write requests.

Request	Count
GET /	110
GET /basket.html	111
DELETE /cart	110
POST /cart	111
GET /catalogue	117
GET /category.html	111
GET /detail.html	111
GET /login	111
POST /orders	117

Read-only requests. We performed tests using workloads consisting only of read requests to quantify the expected response time reductions for applications with read-only workloads when deployed on MEC infrastructures. For this purpose, we modified the load-test service so that it only sent read requests to Sock Shop.

Two types of read requests were used in these experiments: *Get Basket* and *Get Category*. *Get Basket* requests require data to be read from cartDB, which is built using MongoDB. Conversely, *Get Category* requests require data to be read from categoryDB, which is built with MySQL. The load-test service generated workloads with 1000 requests of the appropriate type, as shown in Table 5.

Figure 5 shows that Sock Shop achieved the best average response times for both request types (15 ms for *Get Category* and 38 ms for *Get Basket*) under the *Mostly Edge* scenario. The response times for these two read-only request types were significantly lower than those for the *Cloud Only* scenario – by 30 ms and 31 ms, respectively. This is as expected because in the *Mostly Edge* deployment, all user services and replicas of the database services are hosted at the edge location, so the network delay between the end-user and

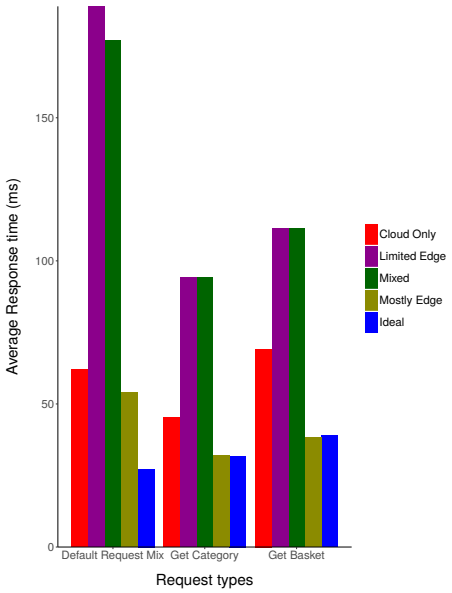


Figure 5: End-to-end Sock Shop response times in different scenarios.

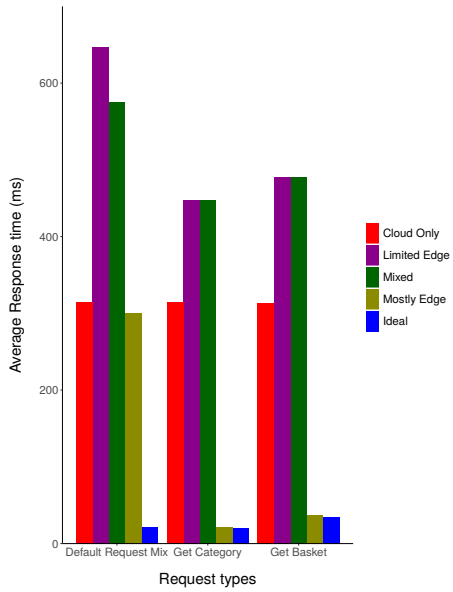


Figure 6: End-to-end response time of Sock Shop in different scenarios (with the latency between the edge datacenter and centralized datacenter is increased to 280 ms).

Table 5: Different types of read-only request delivered to Sock Shop. (Total requests: 1000, concurrent users: 10).

Request	Count
GET /basket.html	1000
GET /catalogue	1000

the application is reduced to 10 ms, whereas the delay time in the *Cloud Only* deployment is 40 ms.

When responding to *Get Basket* and *Get Category* requests, Sock Shop only calls front-end, catalogue, cart, and the corresponding database services. These services are deployed with the same configurations in both the *Limited Edge* and *Mixed* scenarios (i.e., in both cases, the user services are at the edge location, while the database services are in the remote cloud). Therefore, the response times for these read-only request types are identical in these two deployment scenarios, and are substantially higher than those for the *Cloud Only* scenario – 94 ms for *Get Category* and 111 ms for *Get Basket*. Similarly, the Sock Shop response times for the *Mostly Edge* scenario are identical to those for the *Ideal* scenario.

3.6 Latency impact of increased network delay between the edge and the central datacenter

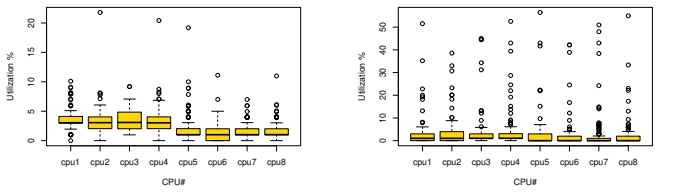
The network parameters of the emulated MECs were initially chosen to reflect latencies typical of locations with high Internet penetration, such as Europe. In regions with lower Internet penetration, such as Africa and South America, the inter-country delay is much higher [24]. To assess the impact of latency on the benchmark applications, we conducted additional experiments using an MEC configuration more representative of such regions, with a single centralized datacenter and edge datacenters scattered around the continent. We set the network RTT delay between the edge location and the centralized datacenter to 280 ms, which is equal to the inter-country delay determined in an earlier study [24].

Figure 6 shows the average response time of SockShop under these conditions. The pattern observed in the previous MEC configuration is reproduced: the *Mostly Edge* deployment achieves a better average response time than *Cloud Only* deployment for both the *default mix* and *read only* request patterns. The response times under *Ideal* scenarios for the *default mix* are again significantly better than those for the *Mostly Edge* deployment, confirming the hypothesis that the delay between the edge and central locations significantly affects application response times.

3.7 Validity of results

To verify that the measured trends in application response times are attributable to network latency, we measured the utilization of the running machine used in the experiments while they were in progress. As presented in section 3.1, we performed all experiments on a local machine with 8 CPU cores. During the running of each experiment(which was approximately 3 minutes from beginning to end under the conditions specified in section 3.4 and section 3.5), we recorded the utilization of each CPU core at 1-second intervals using sar tool[1]. Figure 7a and 7b show the CPU utilization observed when running SockShop and Web Serving, respectively. The utilization of

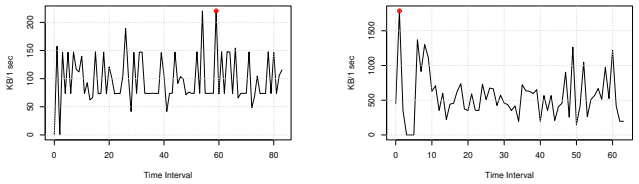
[1]https://linux.die.net/man/1/sar



(a) CPU Utilization in the experiment with Sockshop

(b) CPU utilization in the experiment with Web Serving

Figure 7: CPU utilization measured in during the experiments. The yellow boxes extend from the 25th percentile to the 75th percentile of the CPU utilization dataset, and the thick horizontal line inside each box indicates the median value. The whiskers above and below the boxes show the corresponding maxima and minima, and bubbles show the outliers.



(a) Transmission rate observed in Sockshop's Front-end

(b) Transmission rate observed in Web Serving's web server

Figure 8: Transmission rate observed in application's services (show two services with a detected maximum transmission rate).

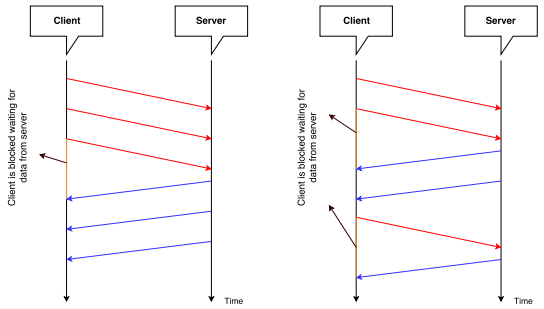
each CPU core was under 10% after eliminating outliers[2], so CPU was not a bottleneck.

We also investigated the extent to which bandwidth constraints affected the final results obtained in the SockShop and Web Serving experiments. validate whether the bandwidth constraints impact the final results in both the SockShop and Web Serving experiments. To this end, we measured the transmission rate (i.e., in KB/second) in and out of each application service. The maximum transmission rate observed in all experiments for SockShop was around 220 KB/second (see Figure 8a), while that for Web Serving was approximately 1,785 KB/second (see Figure 8b). These rates are much lower than the bandwidth assigned to the emulated MECs (1024 Mbps). We are therefore confident that the observed trends are solely due to the emulated network latency.

3.8 Summary

The results presented above clearly show that deploying the cloud-native benchmark applications on MECs only provided benefits

[2]An outlier is an observation that is 1.5 times the interquartile range above the upper quartile or below the lower quartile. At most 8% and 10% of the observations were outliers at any of the eight CPUs for Sock Shop and for Web Serving, respectively.



(a) 1 turn

(b) 3 turns

Figure 9: An example of two sessions in which six messages are passed between two hosts but required different numbers of turns. In Fig. 9a, only one change in the direction occurred in the session, so the number of turns is 1. In Fig. 9b, three changes in the direction occurred, so the number of turns is 3.

that would improve response times in a small number of cases. Moreover, the few improvements achieved by using an MEC instead of deploying exclusively on remote cloud resources were generally not significant.

The largest improvements were observed for read requests when the secondary databases were deployed in edge locations. However, the tasks of cloud-native applications may span diverse categories that involve both write and read requests, such as email, file sharing, customer relationship management, financial accounting, and so on. In the following section, we analyze the architecture and characteristics of these cloud applications to characterize the barriers that prevent them from benefiting from deployment on MECs and thus hinder MEC adoption.

4 UNDERSTANDING LATENCY IMPACT OF DEPLOYMENTS

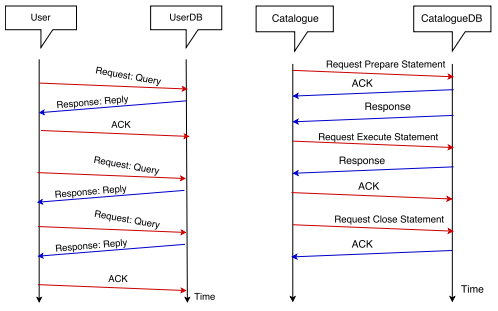
In this section, we first describe our profiler, which is used to explain why current cloud applications derive little benefit in terms of latency reduction when deployed on MECs. We then discuss some potential improvements to cloud application design that would permit such applications to take advantage of MECs.

4.1 Profiling results

We define a **turn** as a change in the direction of communication between two hosts. The number of turns is related to the amount of time in which the client process is blocked, awaiting data returned from the server process. Consequently, the more turns required to service a request, the greater the blocked time.

Figure 9, depicts the flow of communication in two sessions involving the same number of messages but forming different numbers of turns. In the case shown in Figure 9a, there is only one change of direction, so the number of turns is 1. Conversely, the case shown in Figure 9b features three changes of direction, so the number of turns is 3.

Figure 10 depicts the communication in Sock Shop between the hosts of user services and database services using the Mongo and MySQL protocols, respectively. Figure 10a shows that 6 turns are needed to respond to a *Get Login* request in the Mongo protocol session. Conversely, 5 turns are needed to respond to the *Get Catalogue* request in the MySQL protocol session, as shown in Figure 10b.



(a) MongoDB Protocol **(b) MySQL protocol**

Figure 10: The communications in Sock Shop between the two hosts of user services and database services (an extracted TCP messages).

To identify the origins of the high application latency observed for MEC deployments, the profiler needed to measure the number of turns that occurred between two services when servicing specific end-user request types. For this purpose, we used *tcpdump* [25] to capture descriptive information about packets being transmitted and received by container services over the network interface while the benchmark applications were running. We then used the *pypcapfile* package [26], a Python implementation of *libpcap*, to develop a profiling tool that parses and extracts specific information from the collected data to measure turn numbers for different application services.

Figure 11 shows the average numbers of turns between Web Serving services, while Figure 12 shows the total number of turns between Sock Shop services when these two applications respond to the default request mixes described in Section 3.

Notably, **many turns occurred between user services and database services**. For example, Web Serving requires 31 turns on average between web-server and db-server to respond to a request. Similarly, in Sock Shop, the average number of turns between user services and database services is approximately at least 1.6 times

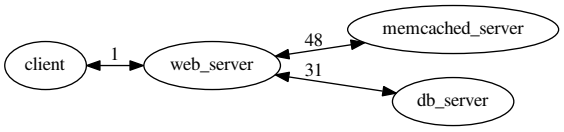


Figure 11: Average number of turns between Web Serving services.

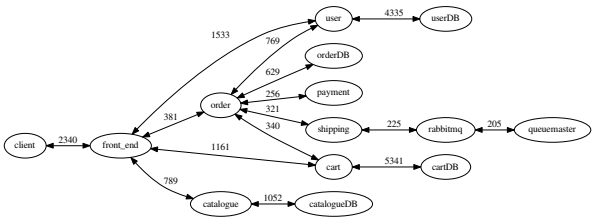


Figure 12: Number of turns between Sock Shop services.

greater than the average number of turns between user-services, or between the end-user and user-services. The number of turns is independent of the deployment scenario, but the delay increases linearly with the distance between network layers. Consequently, the response times of these two applications are significantly increased when user services and database services are placed in different network layers, edge locations and the remote centralized cloud, as observed in the *Mixed* deployment scenario.

In the *Mostly Edge* deployment scenario, the number of end-user requests to the database services on the distant centralized cloud is reduced, so the applications' response times improved compared to the *Cloud only* deployment. However, this improvement was not significant because the aggregated response time also depended on the latency that the database servers on the remote cloud required to respond to the write requests. In cases where only read requests were injected to the application, eliminating all turns to the distant database services, response times were greatly reduced, as seen when Sock Shop responded to *Get Basket* and *Get Catalogue* requests.

Our profiling results suggest that current cloud-native applications tend to make many turns between the user services and its corresponding database services, when responding to end-user's requests. Consequently, deploying these services separately in different network layers obviously causes poor application performance. This is an intrinsic problem that restricts the scope for migrating such cloud-native applications to highly distributed environments such as MECs.

However, such large numbers of turns between the services of cloud native application are not strictly necessary, and can be reduced in some situations. For example, Sock Shop needs 73 turns between the end-user and the front-end (i.e., HTTP/TCP turns) to load all the objects (e.g., CSS, Javascript, and images) used to render the homepage in the end-user's browser upon receipt of a *Get Homepage* request. Similarly, Sock Shop makes 5 turns between the *user* service and the *userDB* service to verify the user's credentials and load user's information in order to responding to a *Login* request. In the former case, the unnecessary number of turn could be diminished by considering the types and the total number of objects that are being dealt. Likewise, in the latter case, the number of turns between the user service and database service could be reduced by optimizing the data query. **Turn reduction solutions** and **response time compression techniques** that achieve these goals are detailed below.

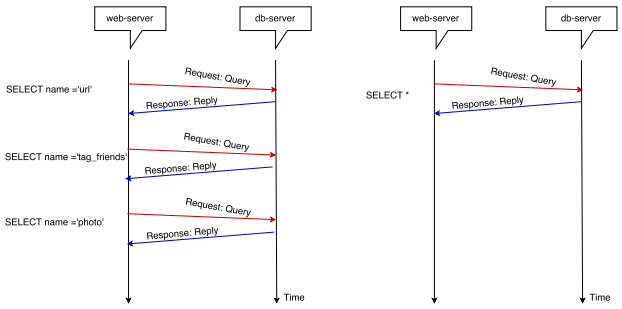
4.2 Recommendations for improving cloud-native application performance on MECs

In this section, we investigated the communication patterns of current cloud-native application architectures to identify potential design improvements that would make it possible to take advantage of MECs. We address the problem at two levels: the application level and the network communication protocol level.

4.2.1 ***Application level.*** The development of cloud applications has prompted the introduction of various innovative solutions to improve Quality of Service (QoS), notably by reducing the need for low latencies between application servers and end-users. Techniques of this sort include *bundling relevant objects, compressing high-bandwidth-requiring objects*, and *caching static content on machines located close to the end-user*. However, these techniques only help improve the latency between the end-user and the application front-end.

To improve the overall performance of cloud applications, it is important to identify approaches that can help reduce response times in the application's back-end between the user services and the corresponding database services.

Query bundling. Microservice-based cloud applications often have private databases for specific services. For instance, the Sock Shop *order service* stores order's records in *orderDB*, while the *catalogue service* stores product information in *catalogueDB*. This one-database-per-service software development pattern requires efficient database design and use: bad designs produce excessive numbers of queries and transactions between the user service and its corresponding database, causing heavy traffic and slow responses if these services are deployed in different network layers. This problem can be alleviated by bundling several queries using *Query join/lookup*. For example, bundling different queries used to retrieve user information in Web Serving makes it possible to retrieve all the user information with just one query, reducing the number of round trips between web-server and data-server. This in turn reduces the response time for Web Serving *login existing user* requests from 4.54 seconds to 3.67 seconds in the *Mixed* deployment.



(a) No query bundling - 5 turns **(b) Query bundling - 1 turn**

Figure 13: An example of query bundling: all the user information is retrieved using a single query instead of many queries.

Caching locally-targeted data at edge locations. Cutting off as many transactions as possible between the user service and the distant database service would improve the application's overall response time. One viable way of doing this is to cache the locally-targeted data at the same locations as the user services, in close proximity to the highest-interest users. This solution reduces the application's response time and also minimizes the central cloud's bandwidth congestion because the end-user requests are dispersed to local services. For example, an application that helps shoppers find the best deals at multiple markets can replicate the data relating to markets close to the end-user's location and deploy them on the server in the end-user's vicinity instead of keeping the whole large database on the distant server.

We quantify the benefit of this technique with the Sock Shop benchmark. In order to reduce the number of transactions to the distant server, a replica of the *cartDB* database is deployed at the edge location along with *carts service* so that the local end-users' selected items are written/read directly to/from this local database. As a result, the average response time for the *check out shopping cart* requests (i.e., writing end user's selected socks to the database *cartDB*) reduces significantly, from 69 ms to 39 ms in the *Mixed* deployment.

Delayed transaction durability/Asynchronous write. As demonstrated by the profiling results presented in Section 3, the response time reductions observed for the *Mostly Edge* deployment are not significant compared to those for the *Cloud Only* deployment. This is because the applications' response times largely reflect the latency of the database services in the remote cloud data center that are needed to respond to write requests. Such delay can be reduced using a technique known as *delayed transaction durability* or *asynchronous write*. Basically, at first the write operations concurrently write new data to a buffer. Whenever the buffer is filled or a buffer flushing event is invoked, then the buffer's data is written to the database. With this technique, the client side does not need to wait for an acknowledgement of the write operation from the server side, no blocking time required at the client side, hence reducing the latency caused by write transactions.

However, this technique leads to a risk of data loss in a catastrophic event (e.g., server crash/shutdown). Therefore, it is important to weigh the trade-off between the performance improvement and the data loss risk. While data loss is undesirable, cloud applications may store various type of data with different importance levels. Social network applications like the Web Serving benchmark is an example, user information is critical hence cannot be lost. However, individual data such as users' posts and messages are not critical. Under some circumstances, it may be worth accepting the potential loss of some of this less important data in order to reduce the latency of the write transactions. The asynchronous write can be applied on database level or transaction level, and is available in both traditional relational database management systems (RDMS) and NoSQL.

We customized the *carts service* of the Sock Shop benchmark so that its write operation is in asynchronous write mode. The number of turns to write one item in *shopping cart* to *cartDB* thus reduces from 15 turns to 11 turns. Therefore, the average response time of Sock Shop for the *check out shopping cart* request reduces

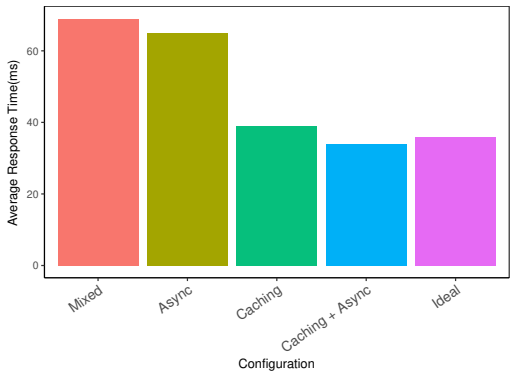


Figure 14: The average response time of the re-designed Sock Shop (i.e., Async, Caching, Caching + Async) to the *check out shopping cart* request as compared against that of the original Sock Shop deployed in the *Mixed* and the *Ideal* scenario.

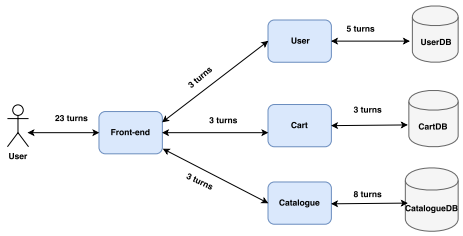


Figure 15: The process of servicing a *Get Homepage* request in Sock Shop.

approximately from 69 ms to 65 ms when deployed in the *Mixed* deployment.

The overall performance of Sock Shop further improves when applying both *Caching locally-targeted data at edge locations* and *Asynchronous write* at the same time. As shown in the Fig. 14, the average response time of Sock Shop to perform the *check out shopping cart* request is around 35 ms which is less than a half as compared to the original Sock Shop in the *Mixed* deployment and approximately equal to that in the *Ideal* deployment (37 ms).

Asynchronous programming. To respond to a user request, many user services and database services command one-another to execute various small independent tasks. These can be executed in parallel. For example, servicing a *Get Homepage* request in Sock Shop requires the involvement of several services, as shown in Figure 15. In the current version of Sock Shop, this occurs synchronously, so the inter-service tasks are executed sequentially. Consequently, the response time for Sock Shop *Get Homepage* requests is the sum of the time spent on all these tasks. However, the tasks associated with the interaction between the *front-end* and *user service* are independent of those resulting from the interaction between the *front-end* and *catalogue* service. Consequently, they can be executed in parallel, potentially improving the overall response time.

4.2.2 ***Network protocol level.*** The above discussion focuses on application-level improvements intended to reduce the number of turns between the services in the back-end of cloud-native applications. We now consider improvements in the network communication protocols used between back-end services that could facilitate the adaptation of cloud-native applications to MECs.

Most connections on the Internet are short transfers and are most heavily impacted by unnecessary handshake round trips [27]. Most modern web-based cloud applications use the HTTP transport protocol over the Internet [28]. HTTP/1.1 allows multiple requests to be pipelined in a single persistent TCP connection without waiting for a response [29], which reduces the number of TCP handshakes and packets needed for transmission across the network. Because HTTP/1.1 retains a first-in-first-out ordering, it can suffer from the head of line blocking problem, which means that if the server takes a long time to respond to a request, subsequent requests must wait. The HTTP/2 [30] protocol was introduced to address this problem. However, the TCP congestion avoidance techniques of HTTP/2 do not fully solve the head of line blocking problem, especially in the event of packet loss.

To tackle this problem, one can employ the Quick UDP Internet Connections (QUIC) protocol developed by Google [27]. QUIC is a reliable multiplexed transport protocol similar to TCP + TLS + HTTP/2, but runs on top of UDP instead of TCP. Briefly, QUIC handshakes frequently require zero roundtrips before sending a payload, as compared to 1 to 3 roundtrips for TCP + TLS. Applications developed using the QUIC protocol have various advantages, including connection establishment latency, improved congestion control, and multiplexing without head of line blocking, thus reducing the number of packets transmitted back and forth across the network.

The QUIC protocol has been widely adopted, especially on the client side. An approximate estimate shows that over 30% of Google's egress traffic and 7% of global Internet traffic uses QUIC as of the time of writing [28]. The latency reductions achieved by deploying QUIC are compelling. For example, Google uses QUIC in the Google Search application, achieving latency reductions of 8% and 3.6%, respectively, for desktop and mobile users [27]. Given the response time reductions that QUIC affords to communications between end-user devices and application servers, deploying QUIC in the application back-end between the user services and database services **could potentially lead to significant improvements in cloud applications' response time**.

5 RELATED WORK

MEC is a paradigm in the early stages of development that has seen limited deployment (e.g., *Lambda@Edge*[3]), many studies have focused on identifying computing infrastructures and application architectures that could be used to realize its potential advantages [1].

Some of the key demonstrators of MECs are latency-sensitive IoT-based applications such as augmented reality and wearable cognitive assistance systems. To evaluate the performance of such applications in terms of end-to-end response times, Zhou et al. [31] conducted empirical experiments using 7 different cognitive assistance applications covering a wide variety of assistant tasks.

[3]https://aws.amazon.com/lambda/edge/

Similarly, Hu et al. [32] examine the benefit of edge in terms of response time and energy consumption by offloading different interactive mobile applications to the edge. Their results showed that the performance of these applications is maximized by offloading to MECs rather than running on a centralized cloud.

Choy et al. [33] helped to increase the maturity of MECs by proposing a hybrid deployment strategy for on-demand game play to meet the strict latency requirements of gaming applications. By using a clever selection mechanism to select the location of a game server at the network edge, these locally-targeted applications can guarantee a certain quality of service even when the number of end-users increases significantly. Lin et al. [34] introduces a lightweight framework for offloading Massively Multiplayer Online Game (MMOG), namely CloudFog, which utilizes resources in the proximity of end-users to relieve the remote cloud's ingress bandwidth, increase the user coverage and reduce response latency.

Real-time video analytics which drive a wide range of novel applications (e.g., surveillance, self-driving car, etc.) are also one of "killer applications" for Mobile Edge Cloud [35]. In order to overcome the problems of high bandwidth consumption, privacy concerns and long latency in the real-time video surveillance system, Tan et al. introduced the Vigil framework [36] which utilize compute resources from the edge of the network. In another spectrum, Karim et al. [37] observed performance gains from edge deployment of a latency-sensitive high bandwidth video analytics application. The authors argued that the benefits of edge deployment become obvious when the computation time is less than the network latency.

In addition to such MEC-native applications, it is important to determine whether MECs can offer any benefits to non-MEC-native applications because failing to do so may hinder the development and adoption of MECs. Among existing application types, cloud-native applications are arguably the best suited for adaptation to MEC platforms, and such adaptation could spur MEC deployment and investment in much the same way that the adoption of traditional clouds was fostered by non-cloud-native legacy applications. However, before moving such applications to MECs, it is necessary to investigate their architecture as well as the techniques being used to develop such cloud applications. For instance in most cloud applications, the Object Relational Mapping (ORM) has been widely employed as a conceptual abstraction to abstract complex database accesses [38]. However, Chen et al. [39] proved that ORM can cause redundant data problems which in turn seriously impacts to the application performance. In recent years, microservice architecture has emerged as a popular framework for engineering cloud applications as its capable of accelerating agile software development, deployment, and maintenance. Some groups have studied the characteristics of microservice application architectures and analyzed their performance when deployed in centralized clouds [40, 41]. On the basis of experiments, they argued that the application implemented in a microservice architecture generates more communication than that in a monolithic architecture, thus diminishes the application's performance in terms of response time. This is again confirmed by results from our work in Section 4.

In the early days of cloud computing, several studies examined the potential benefits and challenges of moving legacy enterprise applications to cloud environments, and the scope for utilizing resources in hybrid environments that combine on-premise and cloud infrastructures. To determine which elements of specific applications should be deployed locally and which should be migrated to cloud datacenters, one must consider the intertwined problems of application complexity (which arises from the interactions between an application's diverse components as well as factors such as privacy considerations) and the variability of application performance in the cloud (which arises from network latency, resource starvation, and so on). Hajjat et al. [42] were among the first researchers to show that significant advantages could be gained by combining on-premise and cloud resources when deploying multi-component applications, and proposed a model that enables automated planning of such cloud migrations. Similarly, Andrikopoulos [43] outlined the challenges of moving legacy applications to the cloud and ways of overcoming these challenges. They also discussed ways of identifying which components of an application can beneficially be migrated and ways of adapting applications for operation in such mixed environments.

Our work complements these earlier studies: we have used two popular cloud-native application benchmarks and exhaustive evaluated their performance (i.e., response times) under various deployment scenarios using combined edge resources and remote cloud datacenter resources. By identifying root-causes of why cloud applications derive little benefit from MECs, we proposed various potential design improvements in software engineering from application level to network communication protocol level, and indeed the quantifying results showed that, cloud applications with such changes are able to amend their overall performance in terms of latency reduction when deployed on MECs.

6 CONCLUSIONS AND FUTURE WORK

MECs are recognized as a key enabling driver of fifth generation mobile network technology (5G) that will make it possible to meet end-user expectations relating to performance, scalability and agility. Various mission-critical application types that are poorly served by current cloud infrastructure could run well on MECs.

Learning from the historical role of non-cloud-native legacy applications in traditional clouds, we argue that MECs also provide benefits to non-MEC-native applications. Therefore, in this work, we conducted empirical studies to explore and quantify the benefits of deploying cloud-native applications on MECs. We deployed two popular microservice benchmarks in different scenarios using resources from edge locations and the remote centralized cloud. Disappointingly, our results showed that current cloud-native applications derive little benefit from deployment on MECs in terms of latency reduction, and may even suffer from increased latency when deployed in this way. We developed a profiler to better understand the causes of these problems, revealing that they originate from the large numbers of transactions between application services when processing end-user requests. The number of transactions multiplied by the network delay between the edge and the remote centralized cloud causes response times to increase dramatically.

We subsequently identified some ways of modifying the engineering of cloud-native applications that could enable them to derive benefits from deployment on MECs. We showed that such changes can bring the performance of a cloud native application

to that expected in an ideal scenario, i.e., in which the latency between the edge location and the remote datacenter has no impact on the application performance. Our paper paves the way to a more rapid adoption of MECs, by enabling a broad class of applications – microservice-based cloud applications – to readily take advantage of MECs.

The study is limited to one edge site and one centralized data centers. However, if applications are deployed in distinct edge sites, the overall result will be relevant those of the *Ideal scenario* plus the delay of data synchronization among edge sites. As near future work, we plan to verify the measurement for this scenario in which many aspects are considered such as database sharding, load balancing, and methods to generate workloads that is able to reflect the interaction of local end-users with applications deployed at the local edge sites. Also, we plan to further investigate the application- and protocol-level improvements discussed herein, with the aim of quantifying the gains and costs associated with each approach. In this way, we will draw up a blueprint for porting cloud-native applications to MECs.

ACKNOWLEDGEMENTS

The authors would like to thank Professor Michael Rabinovich and the anonymous reviewers for their helpful and constructive suggestions that greatly contributed to improving the final version of the paper. Financial support has been provided by the Wallenberg AI, Autonomous Systems and Software Program (WASP) funded by the Knut and Alice Wallenberg Foundation, the Swedish Research Council (VR) under contract number C0590801 for the Cloud Control project, and the Swedish Strategic Research Program eSSENCE.

REFERENCES

[1] H. Liu, F. Eldarrat, H. Alqahtani, A. Reznik, X. de Foy, Y. Zhang, Mobile edge cloud system: Architectures, challenges, and approaches, IEEE Systems Journal 12 (3) (2018) 2495–2508. doi:10.1109/JSYST.2017.2654119.

[2] E. Ahmed, M. H. Rehmani, Mobile edge computing: Opportunities, solutions, and challenges, Future Generation Computer Systems 70 (2017) 59 – 63. doi:https://doi.org/10.1016/j.future.2016.09.015.

[3] F. Khan, The cost of latency, https://www.digitalrealty.com/blog/the-cost-of-latency/, accessed: 2019-04-01 (2015).

[4] Kissmetrics, How loading time affects your bottom line, https://blog.kissmetrics.com/loading-time/, accessed: 2019-04-01 (2011).

[5] J. Lewis, M. Fowler, Microservices - a definition of this new architectural term, https://martinfowler.com/articles/microservices.html, accessed: 2019-04-01 (2014).

[6] S. Newman, Building Microservices, O'Reilly Media, 2015.
URL https://books.google.se/books?id=1uUDoQEACAAJ

[7] A. Vakali, G. Pallis, Content delivery networks: status and trends, IEEE Internet Computing 7 (6) (2003) 68–74. doi:10.1109/MIC.2003.1250586.

[8] G. Pallis, A. Vakali, Insight and perspectives for content delivery networks, Commun. ACM 49 (1) (2006) 101–106. doi:10.1145/1107458.1107462.

[9] Web serving benchmark, http://cloudsuite.ch//pages/benchmarks/webserving/, accessed: 2019-04-01 (2017).

[10] Weaveworks Inc, Socks shop a microservices demo application, https://microservices-demo.github.io, accessed: 2019-04-01 (2016).

[11] P. M. Mell, T. Grance, Sp 800-145. the nist definition of cloud computing, Tech. rep., Gaithersburg, MD, United States (2011).

[12] C. Pautasso, O. Zimmermann, M. Amundsen, J. Lewis, N. Josuttis, Microservices in practice, part 1: Reality check and service design, IEEE Softw. 34 (1) (2017) 91–98. doi:10.1109/MS.2017.24.

[13] Stratoscale, Hybrid cloud survey, https://www.stratoscale.com/solutions/hybrid-cloud/survey/ (2017).

[14] R. Miller, Docker brings containerization to legacy apps, https://techcrunch.com/2017/04/19/docker-announces-new-containerization-service-for-legacy-apps/, accessed: 2019-04-26 (Apr. 2017).

[15] A. Mehta, W. Tärneberg, C. Klein, J. Tordsson, M. Kihl, E. Elmroth, How beneficial are intermediate layer data centers in mobile edge networks?, in: 2016 IEEE 1st International Workshops on Foundations and Applications of Self* Systems (FAS*W), 2016, pp. 222–229. doi:10.1109/FAS-W.2016.55.

[16] I. Hadžić, Y. Abe, H. C. Woithe, Edge computing in the epc: a reality check, in: Proceedings of the Second ACM/IEEE Symposium on Edge Computing, SEC '17, ACM, New York, NY, USA, 2017, pp. 13:1–13:10. doi:10.1145/3132211.3134449.

[17] AWS Regions and Endpoints - Amazon Web Services, https://docs.aws.amazon.com/general/latest/gr/rande.html.

[18] Network emulation, https://wiki.linuxfoundation.org/networking/netem, accessed: 2019-04-01.

[19] Docker repositories, https://hub.docker.com/explore/, accessed: 2019-04-01 (2017).

[20] T. Palit, Y. Shen, M. Ferdman, Demystifying Cloud Benchmarking, in: 2016 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), 2016, p. 122–132.

[21] Faban - Helping measure performance, http://faban.org/, accessed: 2019-04-01.

[22] Overview of docker compose, https://docs.docker.com/compose/overview/, accessed: 2019-04-01 (2017).

[23] Simulate acutual end user usage of sock shop, https://github.com/microservices-demo/load-test, accessed: 2019-04-01.

[24] A. Formoso, J. Chavula, A. Phokeer, A. Sathiaseelan, G. Tyson, Deep diving into africa's inter-country latencies, in: IEEE INFOCOM 2018-IEEE Conference on Computer Communications, IEEE, 2018, pp. 2231–2239.

[25] tcpdump dump traffic on a network, http://www.tcpdump.org/, accessed: 2019-04-01 (February 2017).

[26] Python pypcapfile package, https://pypi.python.org/pypi/pypcapfile, accessed: 2019-04-01.

[27] A. Langley, A. Riddoch, A. Wilk, A. Vicente, C. Krasic, D. Zhang, F. Yang, F. Kouranov, I. Swett, J. Iyengar, J. Bailey, J. Dorfman, J. Roskind, J. Kulik, P. Westin, R. Tenneti, R. Shade, R. Hamilton, V. Vasiliev, W.-T. Chang, Z. Shi, The QUIC Transport Protocol: Design and Internet-Scale Deployment, in: Proceedings of the Conference of the ACM Special Interest Group on Data Communication, SIGCOMM '17, ACM, New York, NY, USA, 2017, pp. 183–196.

[28] Sandvine, Global internet phenomena report (2016).

[29] R. Fielding, J. Reschke, RFC 7230: Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing, Internet Engineering Task Forc (IETF) (2014).

[30] M. Belshe, R. Peon, M. Thomson, RFC 7540: Hypertext Transfer Protocol Version 2 (HTTP/2), Internet Engineering Task Force (IETF) (2015).

[31] Z. Chen, W. Hu, J. Wang, S. Zhao, B. Amos, G. Wu, K. Ha, K. Elgazzar, P. Pillai, R. Klatzky, D. Siewiorek, M. Satyanarayanan, An empirical study of latency in an emerging class of edge computing applications for wearable cognitive assistance, in: The Second ACM/IEEE Symposium on Edge Computing, IEEE, 2017.

[32] W. Hu, Y. Gao, K. Ha, J. Wang, B. Amos, Z. Chen, P. Pillai, M. Satyanarayanan, Quantifying the impact of edge computing on mobile applications, in: Proceedings of the 7th ACM SIGOPS Asia-Pacific Workshop on Systems, APSys '16, ACM, New York, NY, USA, 2016, pp. 5:1–5:8. doi:10.1145/2967360.2967369.

[33] S. Choy, B. Wong, G. Simon, C. Rosenberg, A hybrid edge-cloud architecture for reducing on-demand gaming latency, Multimedia Systems 20 (5) (2014) 503–519.

[34] Y. Lin, H. Shen, CloudFog: Leveraging Fog to Extend Cloud Gaming for Thin-Client MMOG with High Quality of Service, IEEE Trans. Parallel Distrib. Syst. 28 (2) (2017) 431–445. doi:10.1109/TPDS.2016.2563428.

[35] G. Ananthanarayanan, P. Bahl, P. Bodík, K. Chintalapudi, M. Philipose, L. Ravindranath, S. Sinha, Real-time video analytics: The killer app for edge computing, Computer 50 (10) (2017) 58–67. doi:10.1109/MC.2017.3641638.

[36] T. Zhang, A. Chowdhery, P. V. Bahl, K. Jamieson, S. Banerjee, The design and implementation of a wireless video surveillance system, in: Proceedings of the 21st Annual International Conference on Mobile Computing and Networking, MobiCom '15, ACM, New York, NY, USA, 2015, pp. 426–438. doi:10.1145/2789168.2790123.

[37] F. Kalim, S. A. Noghabi, S. Verma, To edge or not to edge?, in: Proceedings of the 2017 Symposium on Cloud Computing, SoCC '17, ACM, New York, NY, USA, 2017, pp. 629–629. doi:10.1145/3127479.3132572.

[38] R. Johnson, J2ee development frameworks, Computer 38 (1) (2005) 107–110. doi:10.1109/MC.2005.22.

[39] T. H. Chen, W. Shang, Z. M. Jiang, A. E. Hassan, M. Nasser, P. Flora, Finding and evaluating the performance impact of redundant data access for applications that are developed using object-relational mapping frameworks, IEEE Transactions on Software Engineering 42 (12) (2016) 1148–1161. doi:10.1109/TSE.2016.2553039.

[40] T. Ueda, T. Nakaike, M. Ohara, Workload characterization for microservices, in: 2016 IEEE International Symposium on Workload Characterization (IISWC), IEEE, 2016, pp. 1–10.

[41] M. Villamizar, O. Garcés, L. Ochoa, H. Castro, L. Salamanca, M. Verano, R. Casallas, S. Gil, C. Valencia, A. Zambrano, M. Lang, Infrastructure cost comparison of running web applications in the cloud using aws lambda and monolithic and microservice architectures, in: 2016 16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid), 2016, pp. 179–182. doi:10.1109/CCGrid.2016.37.

[42] M. Hajjat, X. Sun, Y.-W. E. Sung, D. Maltz, S. Rao, K. Sripanidkulchai, M. Tawarmalani, Cloudward bound: planning for beneficial migration of enterprise applications to the cloud, ACM SIGCOMM Computer Communication Review 41 (4) (2011) 243–254.
[43] V. Andrikopoulos, T. Binz, F. Leymann, S. Strauch, How to adapt applications for the cloud environment, Computing 95 (6) (2013) 493–535.