Cost efficiency under mixed serverless and serverful deployments

Reuter, Anja; Back, Timon; Andrikopoulos, Vasilios

Published in:
Proceedings of the 46th EUROMICRO conference on Software Engineering and Advanced Applications (SEAA '20)

DOI:
10.1109/SEAA51224.2020.00049

Publication date:
2020

Cost efficiency under mixed serverless and serverful deployments

Anja Reuter, Timon Back, and Vasilios Andrikopoulos
Computer Science Department
University of Groningen
Groningen, the Netherlands
Email: a.reuter@rug.nl, timon.back@gmail.com, v.andrikopoulos@rug.nl

***Abstract*—Function as a Service (FaaS) is an integral part of the serverless computing paradigm. It delivers to its adopters a true pay-per-use billing model, and releases developers from the burden of managing the application stack. Existing works have already started the discussion on whether this model is more appropriate for cloud computing users in terms of accruing costs when compared to more "traditional" solutions using the Infrastructure and Platform as a Service delivery models. However, by treating this subject as a regular service selection problem, these approaches fail to exploit the space created by distributing the load between simultaneous FaaS and non-FaaS deployments of an application in a hybrid deployment model. This work aims to provide the means for application owners to not only decide which deployment scenario is cost optimal for their needs, but in case that is a hybrid one, also what is the optimal number of virtual machines that will need to be provisioned. An extensible and configurable FAASSIMULATOR open source tool is presented for these purposes.**

***Index Terms*—serverless, FaaS, cost efficiency, simulation**

I. INTRODUCTION

Serverless computing, the programming model and architecture allowing for code execution in the cloud without control of the resources on which the code runs [1], is becoming increasingly popular. Serverless encompasses a wide range of technologies, but for the purposes of this work we scope the discussion to its *Function as a Service (FaaS)* variant, which allows the execution of business logic directly on a platform offered by a cloud service provider like AWS Lambda[1], Microsoft Azure Functions[2], or Google Functions[3]. Roberts and Chapin [2] identify 5 key benefits that are directly applicable to FaaS: reduced labor cost, reduced risk, reduced resource cost, increased scaling flexibility, and shorter lead time for new products.

Of particular interest and focus for this work are the resource cost reductions. These are accruing from moving the responsibility for the inefficiency-prone operations cycle of resource planning, allocation, and provisioning from the application developer to the cloud service provider. In addition, FaaS service consumers are charged in a model closer to the one used for storage rather than the one that has been used so far for computation solutions like Virtual Machine as a Service (VMaaS) and Container as a Service (CaaS). FaaS billing is pay per use based on the execution time of each function in $100\,ms$ increments, with allocated or consumed memory per function acting as a cost parameter. However, as previously discussed, the pricing model of FaaS solutions can be difficult to decipher and surprisingly complex to model [3] and there are a number of open issues [4] that affect this effort.

In this context, earlier approaches in the literature like [5], [6], and [7] are positioned to solve a binary service selection problem, namely whether it is more cost efficient (i.e. cheaper) to use FaaS or another delivery model like IaaS or PaaS for a given class of applications. The position taken by this paper is that the problem is not necessarily binary, but rather discrete in the spectrum of deployment options which is created by introducing *hybrid deployment scenarios* that mix FaaS and VMaaS. Bursting excess load to public cloud functions from inside a private cloud deployment would be an instance where such an approach is already taken.

In this respect, a *Pareto optimal strategy* which defines the best mix of load between FaaS and VMaaS in terms of resulting cost, instead of a fixed point selection of provider and configuration for a given load might be a better target for cost optimization purposes. This work aims to investigate this proposition and provide evidence towards the cost efficiency of such hybrid, that is mixed FaaS and VMaaS, deployment scenarios for different kinds of load patterns. The realizability and efficacy of this position is already demonstrated by recent works like [8] and [9]. However, in contrast to these approaches, we focus on providing system owners with the means to decide whether and under which conditions such a deployment scenario is appropriate, rather than starting with the assumption that this is so.

Taking the above into consideration, the main contribution of this work is actually two-fold. On one hand, we present a tool simulating the deployment cost across the spectrum of the different deployment scenarios (pure FaaS, pure VMaaS, or different hybrid options) for a number of commonly encountered load patterns. This *deployment cost simulator* is both *extensible* and *configurable*. With respect to the latter point we show how the simulator can be *tuned* to the specific needs and load response of applications through a repeatable process. For the second contribution of this work, we use the simulator to provide insights into how the burstiness of the application load

[1]AWS Lambda: https://aws.amazon.com/lambda/

[2]Microsoft Azure Functions: https://azure.microsoft.com/services/functions/

[3]Google Cloud Functions: https://cloud.google.com/functions/

might especially favor a hybrid deployment in order to produce optimal results in terms of cost minimization, depending on the burst size.

The remaining of this paper is structured as follows. Section II presents the deployment cost simulator in terms of its underlying model, offered features, and options for configuration. Section III discusses the process for tuning it to simulate a desired behavior. We use the tool to estimate the cost of different deployment scenarios for a representative sample function, and discuss our findings in Section IV. Related work is summarized in Section V, and the paper concludes with Section VI discussing also future work.

II. Deployment Cost Simulator

The FaaSSimulator is an open source[4] Python 3 application with an easy to use interactive web interface shown in Fig. 1. After the user defines a specific application load scenario, the tool visualizes the load and automatically calculates the Pareto optimal cost configuration across the deployment spectrum, i.e. from pure VMaaS to pure FaaS and in-between mixed scenarios. In the following we first briefly summarize the basis on which the simulator operates, and then we use the interface shown in Fig. 1 to discuss its features.

A. Operational Principle

Internally, the simulator uses a search-based approach to discover how many resources are required to serve the total amount of requests to a function of interest for a defined number of temporal intervals. In the initial phase of each simulation, the tool calculates the number of incoming (new) and currently under processing requests for each interval, and compares the foreseen application load against the set of existing resources in terms of already provisioned VMs. If it is concluded that the available resources are exceeded, new resources are provisioned by adding more VMs to the set until needs are met. The total amount of requests that each VM can serve simultaneously is assumed to be known and fixed for these purposes, but as we discuss in Section III it can be experimentally estimated. In the initial iteration, no VMs have been provisioned, so the provisioned VMs set is empty. At the end of the initial phase, the total number of VMs required to serve the load has been calculated.

Requests are assigned to the first VM that is able to handle them, that is, it has not reached the a priori known maximum amount of concurrent requests per VM. In many cases this is resulting in VMs that are fully utilized, i.e. having reached their maximum amount of concurrent requests, whereas others are barely used. In the following phases of the simulation, the least utilized VM in terms of assigned requests is removed from the pool, while the requests assigned to it are moved over to FaaS instances. By iterating over the provisioned VMs set, multiple hybrid deployment models varying in the amount of used VMs are created until all VMs are removed. The assumption here is that it is possible to have the same function (code) deployed both in a FaaS service, and as an executable function in a VM. As the related work has demonstrated, e.g. [8], [9], and we discuss further in the following section, this is relatively easily achievable by means of existing tooling.

In each phase, the total execution cost is the sum of individual instance costs for VMs and FaaS instances. Pareto optimality is subsequently identified by the scenario that results into the minimum total cost, and can be easily visualized as discussed next. For a more detailed information about the search strategy and general architecture of the tool, the interested reader is referred to [10].

B. Interface

Starting from the bottom left quadrant of Fig. 1, the FaaSSimulator allows users to define the characteristics of their desired load scenario. The load is characterized as a function of amount of requests per time interval. Each request has a (mean) duration as well as an upper bound of resource consumption, being primary RAM (21 s and 128 MB specifically in the figure). This is necessary due to the FaaS billing model of the big cloud service providers offerings. The user is required to input the mean request duration of the function of interest, as well as the memory usage based on e.g. previous taken use-case dependent measurements (see the following section for more on this matter). Additionally, the total simulation duration in seconds is specified to determine the total simulation length and with it the total amount of simulation steps.

In the same quadrant, the user can also choose a load distribution pattern from a set of available ones, along with the corresponding load distribution parameters. In the example shown in Fig. 1, a normal distribution with a sigma (σ standard deviation) value of 200 related to the total test length of 3600 simulation steps is chosen while simulating a total of 36000 requests. Based on the scenario configuration, the distribution of requests is visualized in the top right quadrant. Defining such a small sigma for the normal distribution results into the spike shown in Fig. 1; a larger sigma value, or a smaller amount of requests in the same length would result into less dramatic load distributions. Modifying the load values allows for exploratory insights and answers to what-if questions. At this point, load models for a constant, normal distribution, sinusoid e.g. representing day-night cycles, sawtooth, box-pattern and triangle load are included in the FaaSSimulator implementation. Nevertheless, custom load models can be added easily by implementing an interface as shown in Listing 1 and registering it in the `Loads` class:

Listing 1. Implementation of the `LoadGenerator` interface to create a constant load

```
class LoadGeneratorConstant(LoadGenerator):
  def get_load_at(self, timestamp: int):
    return self.altitude

  def get_name(self):
    return "constant"
```

[4] FaaSSimulator: https://github.com/search-rug/thesis-msc-simulator

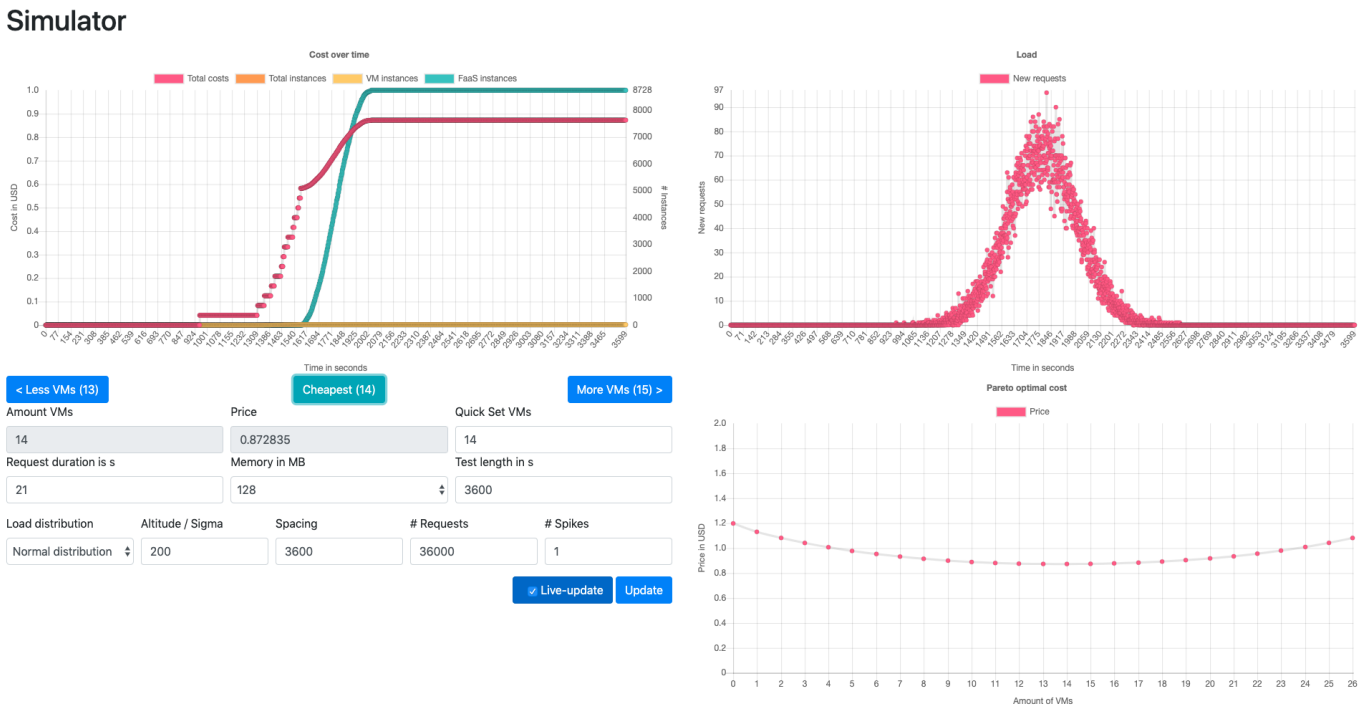


Fig. 1. Screenshot of the FAASSIMULATOR web interface

Replaying a previously recorded or generated load is possible by creating a `LoadGeneratorReplay` instance and passing it a mapping of `simulation_step` to `request_count`.

The top left quadrant of the figure shows the results of the simulation for the provided parameters as a detailed cost graph. It illustrates the accumulated number of started VMaaS and FaaS instances as well as cost in the colours yellow, cyan and magenta, respectively, for a specific deployment scenario. In the particular case of Fig. 1, the calculated Pareto optimal scenario with 14 VMs is chosen. Here, the VMs are able to handle the load for the first third of the simulated load, while continuously more VMs are provisioned as the count of requests per simulation step start ramping up. Based on the simulation outcome, FaaS instances are more efficiently used to handle the bursty spike when it arrives. Notice the rise of the cyan line indicating the use of FaaS instances in the middle of the simulation. Navigation between the different deployment configurations (in terms of provisioned VMs) is realized by clicking the buttons at the bottom of this graph.

Finally, the bottom right quadrant visualizes the cost of the various deployment scenarios as the means for illustrating the Pareto optimal one. The total cost to handle the defined load is shown as the total amount of US dollars (y-axis) for a different number of VMs along the x-axis. In this example, the cost is lowest with 14 VMs, with the remaining requests being handled by FaaS instances in this hybrid deployment model. If only VMs were used instead, 26 instances would be required to handle the same load. As it can be seen in the figure, handling the same load with either a pure FaaS or VMaaS deployment scenario results into suboptimality in terms of cost, with the former being actually slightly more expensive that the latter. If the hybrid scenarios are not taken into account, FAASSIMULATOR can therefore also be used to answer the simpler FaaS vs VMaaS question.

C. Configuration

FAASSIMULATOR provides the option to users to tune it to their particular needs through the `configuration.py` and the `prices.json` file, in addition to what parameters are provided through its web interface. The most relevant parameters of the `configuration.py` file are listed in Listing 2. The first two groups of configuration parameters are defining the timing limits of the simulation and the load distribution to be used, as discussed in Section II-B. In combination with a set of auxiliary parameters (not shown in Listing 2) defining whether to plot the generated data into an image file and where to store the output, these parameters allow the simulator to be run fully automated in script mode, outputting the raw data in CSV files and creating visual plots for different configurations, without the need of interface interaction.

Further settings that are specified in the `configuration.py` file regard the VM and FaaS configurations. These settings provide information about:

- the number of requests that can be handled concurrently by one VM (`vm_parallel`) and the respective expected (mean) execution time per request (`request_duration`),
- the maximum memory (`request_memory`) consumed per request and the (mean) execution time per function

Listing 2. Relevant settings in the configuration file

```
# Timing limits of the simulation
self.simulation_start = 0  # in seconds
self.simulation_end = 3600  # in seconds

# Load configuration
self.load_name = None
self.load_altitude = 1
self.load_spacing = 3600
self.load_num_requests = 36000
self.load_num_spikes = 1

# VM configuration
self.vm_parallel = 175
self.request_duration = 30.575  # in seconds

# FaaS configuration
self.request_memory = 229  # in MB
self.faas_performance = {
  128: 19.6545,
  256: 9.594,
  512: 4.191,
  1024: 2.189,
  2048: 1.078
}

# random request generator
self.request_generator_seed = 10
self.request_variation = 0.0  # Percentage
```

configuration organized by allocated memory size of the configuration (`faas_performance`).

As discussed previously, these parameters can be estimated experimentally through a tuning process which will be explained in detail in the following section. In addition, the configuration parameters `request_generator_seed` and `request_variation` enable the use of a deterministic request duration randomizer on a specific percentage of requests. The idea in this case is to introduce noise in the execution time, closing the gap between the real world and a possibly overfitted simulation. However, this happens at the expense of the repeatability of each simulation.

Last, but not least, the pricing information for different FaaS and VMaaS offerings is read from a separate `prices.json` file, containing a simple lookup table with the pricing models of both FaaS and VMaaS offerings to be used by the simulator. This information can be easily acquired from the Web pages of the public cloud providers. The tool repository contains a sample prices file which was used for the simulation runs discussed in the following.

III. Tuning the Simulator

Listing 2 contains a number of configuration parameters that have to be adapted to the specifics of a simulated system. These concern on the one hand the configuration of the load within the scenario — which can also be defined through the web interface as discussed in the previous section — and on the other hand the performance parameters for FaaS and VMaaS offerings. For this purpose, and since FaaSSimulator is confined at the time being to simulating atomic function executions, a reference function is to be used for the definition of these parameters. For the simulations executed in this paper, we chose Fast Fourier Transformation (FFT) as an example computation with high requirements regarding computation as well as memory consumption which provides interesting insights into how FaaS services are realized [11]. More specifically, we opted for the $O(n \log n)$ Cooley-Tukey-based FFT algorithm as implented by the `fft-js` library[5] (version 0.0.11) of Node.js. Selecting a JavaScript-based implementation allows us to deploy the same codebase in both FaaS and VMaaS solutions with minimum effort by using the `Serverless` framework[6].

TABLE I
Performance measurements on AWS Lambda

Memory size (MB)	Mean response time (ms)
128	19654.5
256	9594
512	4191
1024	2189
2048	1078

To simulate the deployment options for the FFT function, the correct configuration parameters had to be determined by measuring the performance of the algorithm on both FaaS instances as well as VMs. With respect to the former, the parameters to be determined for the function execution on FaaS are the maximum memory consumption of the function (parameter `request_memory` in Listing 2) as well as the performance of the function on the different FaaS size offerings (`faas_performance`). For this paper, the performance measurement of FaaS was achieved by executing the microbenchmark presented in [11], which executes a set of functions on the different sized FaaS offerings of the major cloud providers, including the FFT implementation we chose as a reference function. However, any function from a similar FaaS-specific benchmark like [12]–[16] can be used for this purpose instead. For the configuration of the simulation, the obtained performance measurements for FFT with input parameter 131072 discrete signals as executed on the different sized FaaS offerings of AWS are used as listed in Table I. The maximum memory consumption, as reported by the microbenchmark, was measured at 229 MB.

For the VM configuration part of the simulation, the relevant parameters are the `request_duration` as well as the number of requests that can be handled in parallel by a single instance (`vm_parallel` in Listing 2). In the interest of exploring the effect that the VM offering selection has in the performance of the simulated system we chose to deploy the Node.js FFT implementation in three different VM types on AWS EC2, as shown in Table II. JMeter[7] was used to determine the maximum number of requests that can be handled in parallel by a VM without causing error responses through a simple saturation process by means of a

[5]https://www.npmjs.com/package/fft-j
[6]https://github.com/serverless/serverless
[7]https://jmeter.apache.org

TABLE II
AWS EC2 INSTANCE DETAILS

	t3.medium	t3.large	t3.xlarge
vCPU	2	2	4
Memory (GB)	4	8	16
Price/hour	\$0.0418	\$0.0835	\$0.1670

TABLE III
PERFORMANCE MEASUREMENTS ON AWS EC2

		t3.medium	t3.large	t3.xlarge
Response time (ms)	min	1750	1415	1499
	max	41071	49582	59420
	mean	21413	25456	30575
Number of requests		60	75	175

load ramp-up. More specifically, JMeter was set up to submit an incrementally increasing amount of requests in rounds until the Node.js server responded with error messages. The measurements from the last round without errors are to be used to configure the relevant parameters of the simulator. The ones for the VMs selected in our example are shown in Table III. Listing 2 therefore shows the configuration settings for running a simulation using `t3.xlarge` VMs in combination with AWS Lambda calls where 256 MB are allocated per function.

An interesting observation here is that the measured performance trend in Tables I and III is different for the two deployment scenarios, despite using the same reference function. Doubling the amount of allocated memory in Lambda configurations yields an improvement of factor 2 in terms of response time. On the other hand, moving from a `t3.medium` to a `t3.large` VM yields an improvement of only around 19% with a 25% increase of simultaneous requests supported, despite the doubling of available memory. Moving from `t3.large` to `t3.xlarge`, however, improves performance metrics by 20% and 133%, respectively, showing the sensitivity of the FFT function to the amount of provisioned CPU (virtual) cores. Notice also that the simulated scenarios assume an idealised setting with perfect scaling, that is, a new VM is started exactly before the load exceeds the amount of requests that can be handled by the currently provisioned virtual machines. This results in a scenario where each machine is brought up to maximum utilisation before scaling is applied. In a production deployment scaling would most likely be applied much earlier (e.g. at 80% utilization) by means of an autoscaling solution. However, by adjusting the measurements accordingly, the simulator can also be used to obtain results for those scenarios.

IV. SIMULATIONS

In the following, we are using the FAASSIMULATOR tuned to the configuration parameters defined in the previous section to investigate the deployment scenarios spectrum with respect to optimality. Since FaaS is advertised as clearly beneficial for systems with bursty loads [1], we structure our simulation runs around figuring out the effect of "burstiness" to cost per deployment scenario. In order to determine how bursty the load on the application has to be to make a hybrid deployment or a FaaS deployment more cost efficient than a VM deployment, the load pattern of a normal distribution (as discussed in Section II) was used. A series of simulation runs with different sigma values ($\sigma = 100, 200, 400, 600, 800$) were executed for 36000 requests in total over a duration of 3600 seconds. Table IV summarizes the results of these runs for each chosen VM size (`t3.medium`, `t3.large`, `t3.xlarge`). As it can be seen in the table, a hybrid deployment scenario actually results almost always in less total projected costs for all values of sigma and VM size — the only exception being the very bursty $\sigma = 100$ for `t3.large` and `t3.xlarge` machines where the sheer cost of the necessary VMs drives the optimal to the pure FaaS model. This appears to confirm the position of this paper that hybrid scenarios are actually more cost effective than their respective pure $*$aaS ones, for the majority of cases. What needs to be examined further is how significant are these savings in comparison to pure F- or VMaaS deployments, as discussed in the following.

A. Pure FaaS and pure VMaaS Deployments

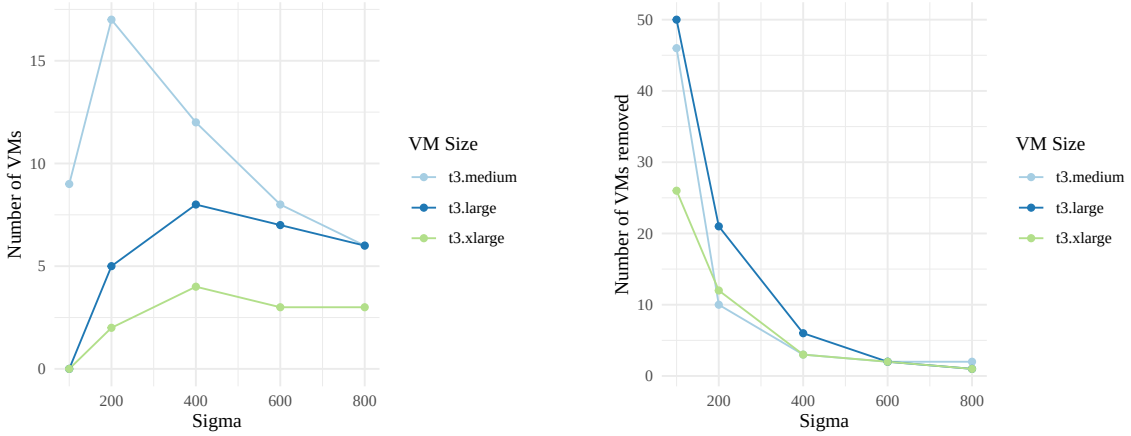
Since the total number of requests stayed the same over all simulation runs, the number of simulated function calls in the pure FaaS deployment is constant, resulting in the same price for all sigma values within the pure FaaS deployment as shown in Table IV. The price for the pure VMaas deployment, however, depends on the capacity and the price of the chosen instances. As shown in Table II, the price per hour for a `t3.medium` machine is about half the price of a `t3.large` which costs half as much as a `t3.xlarge` machine. However, the measurements in Table III show that the capacity of the medium sized VM is rather close to the one of the large machine, while the extra large machine can handle roughly double the amount of requests in the same time frame. This is also reflected in the number of VMs in the pure VMaaS deployment scenario shown in Table IV. The total number of VMs in the pure VMaaS deployment is only slightly higher for medium VMs as they are for large ones, while it takes roughly twice as many large as extra large VMs to execute the same load. This leads to a similar price for deployments on `t3.large` and `t3.xlarge` machines, while the deployments on `t3.medium` machines end up being cheaper. When comparing therefore pure VMaaS and pure FaaS deployment prices, the FaaS deployment is more cost efficient for low sigma values (representing high peaks) while a pure VM deployment is more cost efficient towards high sigma values (representing low peaks). This confirms the position taken by the literature on the subject.

B. Pareto Optimal Deployments

The two graphs in Fig. 2 show the number of VMs in the respective Pareto optimal hybrid deployments for each sigma value, and the difference in the number of VMs in that hybrid deployment versus the amount of VMs needed in a pure VMaaS deployment. More specifically, in Fig. 2a,

TABLE IV
COSTS FOR PURE VMAAS AND FAAS DEPLOYMENTS, AND CORRESPONDING PARETO OPTIMAL HYBRID DEPLOYMENT.

		pure VMaaS		pure FaaS	Pareto optimal hybrid	
VM size	sigma	#VMs	Cost (US$)	Cost (US$)	#VMs	Cost (US$)
t3.medium	100	55	2.2880	1.441152	9	1.360468224
	200	27	1.1232		17	0.951875584
	400	15	0.6240		12	0.533227200
	600	10	0.4160		8	0.377355616
	800	8	0.3328		6	0.290672832
t3.large	100	50	4.1600	1.441152	0	1.441152000
	200	26	2.1632		5	1.335815264
	400	14	1.1648		8	0.911116256
	600	9	0.7488		7	0.656779456
	800	7	0.5824		6	0.517894944
t3.xlarge	100	26	4.3264	1.441152	0	1.441152000
	200	14	2.3296		2	1.346610400
	400	7	1.1648		4	0.928450112
	600	5	0.8320		3	0.685388832
	800	4	0.6656		3	0.521537856



(a) Number of VMs in the Pareto optimal hybrid deployment.

(b) Difference of the number of VMs in the Pareto optimal hybrid deployment vs. pure VMaaS deployment.

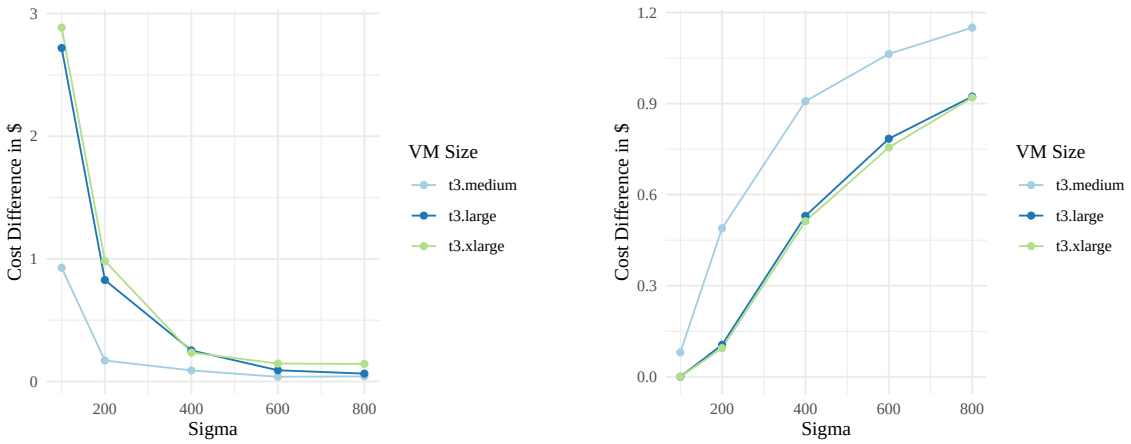
Fig. 2. Development of the number of VMs in Pareto optimal hybrid deployments for different sigma values.

the curves for the number of VMs in the Pareto optimal hybrid deployment are similar between large and extra large VMs. This relates to the aforementioned correlation between the price and capacity of `t3.xlarge` against those by the `t3.large` machine since the price per hour doubles from large to extra large, and roughly twice the amount of VMs are needed for a pure VMaaS deployment. However, the curve for `t3.medium` VMs differs significantly due the price for medium VMs being half of the price for large ones, while only slightly less VMs are needed for a pure VM deployment. For higher sigma values (and therefore less bursty load), the curve for `t3.medium` converges towards the curve for `t3.large`, since the Pareto optimal deployment for higher sigma values converges towards a pure VMaaS deployment. Looking at the difference in the number of VMs in the Pareto optimal hybrid deployment versus the pure VMaaS deployment in Fig. 2b, similar curves appear for large and extra large VMs while the medium sized VM deployment scenario shows a steeper slope due to the previously mentioned difference in the price-performance ratio.

Comparing now the estimated execution costs for the various deployment scenarios, Fig. 3a clearly shows that the highest potential cost savings through a Pareto optimal hybrid deployment with respect to a pure VMaaS deployment are achieved for high peaks in load (low sigma values), while only small gains are achieved for higher sigma values. In contrast, compared to a pure FaaS deployment, the highest potential cost savings through a hybrid deployment are actually achieved towards large sigma values as shown in Fig. 3b.

C. Discussion

The presented simulation results demonstrate that a hybrid deployment model can provide the opportunity for cost savings in scenarios with peaks in the load when compared to either a pure FaaS or a pure VMaaS deployment. While a



(a) Cost difference of the Pareto optimal hybrid deployment vs. pure VMaaS deployment.

(b) Cost difference of the Pareto optimal hybrid deployment vs. pure FaaS deployment.

Fig. 3. Cost differences in US$ across the different deployment scenarios.

hybrid deployment is much more cost efficient than a VMaaS deployment for scenarios with very bursty load patterns, the cost savings for scenarios with lower peaks are limited. In the same manner, a hybrid deployment can be more cost effective than a pure FaaS deployment for scenarios with smaller load peaks, but for very high load peaks a pure FaaS deployment can actually be the Pareto optimal solution. The optimal deployment scenario is therefore highly dependent on the expected load pattern.

The option of a hybrid deployment requires the function to be developed in a way that it can be executed both on FaaS and on VMaaS and an appropriate load balancer has to be in place, as discussed for example in [8]. Running the respective simulation in the FAASSIMULATOR can provide an insight into the possible cost savings which can facilitate the decision if a hybrid deployment is worth the additional effort of implementing the hybrid deployment option, or if a pure VMaaS or pure FaaS deployment is the better choice. In order to provide a meaningful simulation, the tuning procedure needs to be close to the real environment; as seen in the difference between the costs for the `t3.medium` and the `t3.large` machines, the cost difference between different kinds of VMaaS offerings can also make a huge difference for a specific scenario. A related threat to the validity of this work is the lack of a verification and validation process as the one discussed in [17] which could provide some evidence towards its fitness to the "real world" (since actually proving the accuracy is impossible for all possible scenarios). This threat is currently mitigated by the same tuning process which configures the simulator as close as possible to the experimentally observed behavior of the deployed functions, but nevertheless requires further steps to be taken in the future.

The major limitation of our work is that the FAASSIMULATOR is only taking a single function into account and does not incorporate the option of possible function compositions that are common for developing serverless applications in practice. Even though compositions can currently be approximated by means of simulating each function separately and aggregating the results, this approach could fail to consider (network) latencies when shifting between FaaS and VMaaS deployments. An approach similar to the one presented in [18] for the modelling of function compositions based on machine learning can be used for this purpose. It also relies on a rather simple pricing model relying on billing per hour for VMs and without monthly discounts for function executions. These limitations are to be addressed in future work.

V. RELATED WORK

Related work can be roughly distinguished into two groups. In the first group are works that aim to support system owners in deciding the optimal service delivery model for their needs considering the trade-off between cost and quality of service. For example, Villamizar et al. [7] and Albuquerque Jr. et al. [5] combine performance measurements with analytical models to compare (pure) FaaS deployments with equivalent IaaS- and PaaS-only deployments, respectively. The CostHat model for microservices-based applications is used for the same purpose in [19]. The approach for cost prediction of serverless workflows presented in [18] provides a fairly accurate mechanism for predicting the cost of applications as function compositions, which can be used for deciding whether a serverless deployment scenario is profitable for a given application in comparison to a VM-based deployment.

More closely to our work, Boza et al. [6] presented CLOUDCAL, a simulation based optimizer (similar in spirit to our FAASSIMULATOR) which incorporates different pricing models for VMaaS (on-demand, reserved, bid), and FaaS. Given a configurable queue theory model of the to-be deployed system, CLOUDCAL allows for identifying the optimal solution in selecting between reserved VMs, on-demand VMs, and pure FaaS deployments while preserving user-defined Service Level Objectives (SLOs). As discussed in the introductory section,

such works fail to exploit the space created by combining VMaaS with FaaS solutions that can yield significant cost savings with respect to pure deployments as we have shown in the previous section.

The second group of works is concerned with applications deployed simultaneously in both VMaaS and FaaS solutions. These approaches use machine learning techniques to monitor the incoming load and performance behavior of a target application in order to identify where to route the call to (i.e. in the VMaaS or FaaS deployment of the application) for optimal cost/performance results. Horovitz et al. [9] and Gunasekaran et al. [8] are examples of such approaches. The models of hybrid deployment optimization are definitely more sophisticated in these approaches in comparison to ours, and allow for dynamic response to system load changes. However, both works come short in discussing the overhead of their proposed solution for both learning and operation. Furthermore, they do not provide any a priori insights to system owners about projected costs, or any indication with respect to the amount of VMs to be used e.g. for configuring the autoscaling mechanism.

VI. Conclusion

This paper started with the proposition that a hybrid deployment scenario which distributes the incoming system load between deployments in Function as a Service and Virtual Machine as a Service delivery models is in many cases more cost efficient than a pure FaaS or VMaaS scenario. Recent works from the literature are developing machine learning-based approaches for this purpose and try to discover the optimal mix automatically. In our work we aim to develop the means for exploring the deployment spectrum in order to allow system owners to decide which scenario works better for them. For this purpose we developed FaaSSimulator, a configurable and extensible tool which simulates a user-provided request load to a reference function deployed in both FaaS and VMaaS offerings, and calculates the projected cost under different load distribution scenarios. We then used it to show in practice how the type and size of a burst in the system load may require very different deployment scenarios when cost efficiency is considered. Hybrid deployment scenarios turned out to be Pareto optimal in the majority of the simulated cases we examined.

In terms of future work, there are a few directions that can be pursued based on these outcomes and the discussion in Section IV. The first one is to develop more fine-grained pricing models, incorporating for example discounts for number of function invocations per month and the option for more complex pricing models like AWS charging per second after the first minute of VM provisioning for some VM types. The second one is the development of a toolkit which automates the tuning process discussed in Section III, or at least minimizes human input into it. Adding the option for Container as a Service deployments e.g. in AWS ECS[8] would also be an interesting feature. The more important functionality to be added to FaaSSimulator, however, is the option to combine multiple functions under one application profile.

[8]AWS Elastic Container Service: https://aws.amazon.com/ecs/

Acknowledgements

The work presented in this paper has been partially funded through the ITEA 3 project VISDOM: Visual diagnosis for DevOps software development.

References

[1] I.Baldini *et al.*, "Serverless computing: Current trends and open problems," in *Research Advances in Cloud Computing*. Springer, 2017, pp. 1–20.

[2] M.Roberts and J.Chapin, *What is Serverless?* O'Reilly Media, 2017.

[3] A.Eivy, "Be wary of the economics of "serverless" cloud computing," *IEEE Cloud Computing*, vol. 4, no. 2, pp. 6–12, 2017.

[4] L.Wang *et al.*, "Peeking behind the curtains of serverless platforms," in *2018 USENIX Annual Technical Conference (USENIX ATC18)*, 2018, pp. 133–146.

[5] L. F.Albuquerque Jr *et al.*, "Function-as-a-service x platform-as-a-service: Towards a comparative study on faas and paas," in *Proceedings of ICSEA 2017*. IARIA, 2017, pp. 206–212.

[6] E. F.Boza *et al.*, "Reserved, on demand or serverless: Model-based simulations for cloud budget planning," in *2017 IEEE Second Ecuador Technical Chapters Meeting (ETCM)*, Oct 2017, pp. 1–6.

[7] M.Villamizar *et al.*, "Infrastructure cost comparison of running web applications in the cloud using aws lambda and monolithic and microservice architectures," in *16th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid)*. IEEE, May 2016, pp. 179–182.

[8] J. R.Gunasekaran *et al.*, "Spock: Exploiting serverless functions for slo and cost aware resource procurement in public cloud," in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*, July 2019, pp. 199–208.

[9] S.Horovitz *et al.*, "Faastest - machine learning based cost and performance faas optimization," in *Economics of Grids, Clouds, Systems, and Services*, M.Coppola *et al.*, Eds. Cham: Springer International Publishing, 2019, pp. 171–186.

[10] T.Back, "Hybrid serverless and virtual machine deployment model for cost minimization of cloud applications," Master's thesis, University of Groningen, Groningen, The Netherlands, 8 2018. [Online]. Available: https://fse.studenttheses.ub.rug.nl/18484/

[11] T.Back and V.Andrikopoulos, "Using a microbenchmark to compare function as a service solutions," in *Service-Oriented and Cloud Computing*, ser. Lecture Notes in Computer Science, K.Kritikos, P.Plebani, and F.De Paoli, Eds., no. 11116. Cham: Springer, 2018, pp. 146–160.

[12] M.Malawski *et al.*, "Benchmarking heterogeneous cloud functions," in *Euro-Par 2017: Parallel Processing Workshop*. Springer, 2017, pp. 415–426.

[13] W.Lloyd *et al.*, "Serverless computing: An investigation of factors influencing microservice performance," in *Proceedings of the IEEE International Conference on Cloud Engineering (IC2E 2018)*. IEEE.

[14] H.Lee, K.Satyam, and G.Fox, "Evaluation of production serverless computing environments," Tech. Rep., 04 2018. [Online]. Available: http://dx.doi.org/10.13140/RG.2.2.28642.84165

[15] R.Pellegrini, I.Ivkic, and M.Tauber, "Function-as-a-service benchmarking framework," *arXiv preprint arXiv:1905.11707*, 2019.

[16] J.Kim and K.Lee, "Functionbench: A suite of workloads for serverless cloud function service," in *2019 IEEE 12th International Conference on Cloud Computing (CLOUD)*. IEEE, 2019, pp. 502–504.

[17] M.Zakarya and L.Gillam, "Modelling resource heterogeneities in cloud simulations and quantifying their accuracy," *Simulation Modelling Practice and Theory*, vol. 94, pp. 43 – 65, 2019.

[18] S.Eismann *et al.*, "Predicting the costs of serverless workflows," in *Proceedings of ICPE'20 (to appear)*. IEEE.

[19] P.Leitner, J.Cito, and E.Stöckli, "Modelling and managing deployment costs of microservice-based cloud applications," in *Proc. IEEE/ACM 9th Int. Conf. Utility and Cloud Computing (UCC)*, Dec. 2016, pp. 165–174.