Cloud Functions for Fast and Robust Resource Auto-Scaling

Joe H. Novak, Sneha Kumar Kasera, Ryan Stutsman
University of Utah

***Abstract*—We design and build FEAT, a new scaling approach that uses (1) *cloud functions* as interim processing resources to compensate for VM launch delays and (2) a reactive, knobless, auto-scaling algorithm that requires *no* pre-specified thresholds or parameters, making it robust against changing load. We implement FEAT on Amazon Web Services (AWS) and Microsoft Azure. Our evaluations clearly demonstrate the higher performance and robustness of FEAT in comparison to existing approaches.**

I. Introduction

A tenant buys services from a public cloud provider to host applications on virtual machines (VMs). The tenant sells services of the applications to its clients. Cloud providers offer the tenant the ability to automatically scale (auto-scale) applications to meet ever-changing processing demands.[1] However, two critical problems must be addressed to effectively auto-scale resources. First, VMs can take tens of seconds or even minutes to launch [16], and more time is needed to load the application on the VM. During this time, demand on the existing VMs can exceed capacity, resulting in increased response times. Second, determination of the processing power to add or remove is tricky as application load can vary unpredictably over different time scales. In existing scaling approaches, tenants must pre-specify thresholds to cloud providers to trigger scaling operations, but these thresholds are not robust against changing load demands. We design and build a novel scaling approach, FEAT (*F*ast launch *E*vent-driven *A*uto-*T*uning) that uses (1) *cloud functions* (CFs) available at cloud providers as interim resources to deal with the delay in launching VMs and (2) a reactive, knobless, auto-scaling algorithm that does not require any pre-specified thresholds, making it robust against changing load. Further, shifting workload between CFs and VMs may lead to a new programming paradigm to take advantage of the unique characteristics of the respective computing resources.

Provisioning compute in cloud services is undergoing a transition as developers begin to deeply incorporate CFs into their applications. All major cloud providers now offer CFs, including Amazon Web Services (AWS) Lambda [1], Microsoft Azure Functions [17], and IBM OpenWhisk [11]. Tenants benefit from CFs, since they provide low-latency launch times and are inexpensive for short-lived requests. Cloud providers benefit, since CFs are generally small, short, and stateless, which eases resource scheduling and reduces idle capacity. However, CFs have two serious drawbacks compared to VMs. First, they break the classic OS process abstraction, so applications must be reworked to take advantage of them. Second, their pricing model differs from VMs; certain lightweight workloads are more cost effective when run with CFs, whereas heavier workloads that can amortize costs are significantly more cost effective on VMs. This leaves developers with hard choices. For certain operating points, CFs are cost effective and scale more rapidly; for other workloads VMs are cheaper. Worse, developers cannot fluidly trade between these resources as their workload shifts and scales. FEAT uses *heavy CFs* that adapt CFs to work with applications developed for conventional VMs without requiring code changes, and a *knobless* reactive scaling algorithm that employs these heavy CFs to cut costs. At a high level, FEAT keeps a pool of CFs to temporarily service requests when load grows and allocates them based on changes in queuing delay. To keep CF costs from growing, it allocates VMs and shifts load to the VMs as they slowly come online, at which point CFs are idled. An idle CF is about 100× cheaper than an idle VM (§IV), and given that these CFs eliminate VM over-provisioning for scale out, the savings are significant.

Despite differences in programming model, providers' isolation of CFs is not fundamentally different from more conventional approaches, so heavy CFs can be implemented atop existing cloud provider CF offerings. We implement and evaluate heavy CFs as part of FEAT on both AWS and Azure. FEAT not only improves cost savings over state-of-the-art auto-scaling approaches, but also significantly simplifies development and operational management of applications. Developers need not write code specialized for VMs or CFs; both run conventional application code designed for ordinary VM deployment. Developers also need not pre-specify any parameters or thresholds; FEAT automatically provisions CFs and VMs to adapt to changing load while minimizing cost.

We demonstrate FEAT's promise with two common classes of cloud applications: publisher/subscriber and request/response-based applications. Importantly, we compare FEAT against state-of-the-art auto-scaling frameworks (including combined predictive, reactive, and machine learning-based approaches) and show its simple, knobless, reactive approach supported by CFs reduces tenant-observed queuing delay by 40% in our experiments. Interestingly, even state-of-the-art baseline approaches can be improved with FEAT's heavy CFs. FEAT is the first approach using CFs with conventional application code to quickly add processing power when scaling out. Evaluating our implementation on AWS and

[1]Horizontal scaling adds/removes VMs (*scale out*/*scale in*). Vertical (*scale up*/*scale down*) adds/removes resources. We use scale in and scale out for both.

Azure with both synthetic data sets and real-world network traces, we make the following significant observations:

- FEAT's CFs reduce latency up to 2× compared to approaches that do not use CFs, even under 10× change in offered load. It reduces core count over cloud provider scaling by 15-20%.
- FEAT's parameterless approach reduces queuing delay over existing approaches that require tenants to pre-specify thresholds by 5× and existing state-of-the-art approaches by 40%.
- We find AWS has faster VM launch times and less restrictive security in the CF execution environment than Azure. Consequently, our CF method has greater benefit on Azure.
- We find that without CFs, horizontal scaling can attain about 25% lower queuing delay than vertical scaling. Our use of CFs tends to equalize the two.

We build a discrete event simulator to examine the stability of FEAT. We find that FEAT maintains a stable queuing delay in all cases that we study. In summary, we present a new cloud resource scaling architecture and implement it on two cloud provider platforms. Our evaluation shows FEAT's higher performance and robustness compared to existing approaches.

II. Related Work

Since major cloud providers began offering CFs or "lambdas," researchers have been investigating these platforms themselves [9] as well as CF-based applications. Commonly, CFs are used for distributed systems like compute platforms [13] or message brokers [18]. FEAT's use of CFs to temporarily absorb increases in load is new, but others have used them to run specific user-level executables; for example, distributed video processing [6], firewalls and intrusion detection [20]. Reactive load control auto-scaling models scale in response to quantities measured in real time such as CPU utilization or request rate. Predictive models use past workload training sets in machine learning (ML) to attempt to predict future resource demands. Hybrid models combine these approaches. Unlike our approach, most require pre-specification of parameters and thresholds.

Reactive: Cloud providers offer rule-based auto-scaling. When a metric such as CPU utilization exceeds a threshold, a tenant-defined rule fires to add VMs. We compare FEAT with this baseline algorithm as well as a hybrid approach, described below. Gandhi et al. [7] concentrate requests to a set of servers. They use idle timers to determine when to scale in remaining servers. They infer the capacity of a VM and measure workload to scale out. Lim et al. implement feedback controllers that track CPU utilization [14] and target storage tiers [15].

Predictive: Tesauro et al. [21] use reinforcement learning to provision, but it must be trained offline, since training is costly. Nguyen et al. [19] use wavelets to determine resource demand. They clone VMs to mitigate launch delays. We use CFs to handle additional load until VMs have launched.

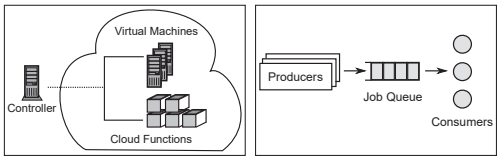
Hybrid: Wang et al. [24] combine reactive, predictive, and feedback control. Their system uses arrival rate to predict the number of cores and fine-tunes them with a latency feedback signal. We implement this existing state-of-the-art algorithm for comparison in our evaluation. The reactive component of Jiang et al. [12] triggers when the request queue exceeds a threshold. It solves a cost/latency trade-off optimization with exhaustive search that does not necessarily scale. Parameters of the reactive component of Urgaonkar et al. [22] include the quantity of work performed per request and an execution frequency.

III. Scaling Architecture

A major issue with scaling VMs is that they are slow to launch, on the order of minutes [16]. One way to deal with this is to keep spare VMs idle, but this is expensive and inefficient. Providers charge for VM run time whether it is performing useful work or sitting idle. Further, it is difficult to predict how many spare VMs will be required. On the other hand, CFs have fast sub-second launch times and are inexpensive to idle. Providers charge CFs by the request and not for idle time. It costs a tenant two orders of magnitude more to idle a spare VM compared to a similarly provisioned CF.

CFs have been more resource limited than VMs, but trends are driving them closer to one another. The memory-to-core ratio in a VM is typically between 2 to 4 GB/core. CFs are quickly approaching this ratio. AWS recently [2] doubled the available CF memory from 1.5 GB to 3 GB. We suspect this trend will continue with increasing demand for CFs and from competition between cloud providers, allowing CFs to run heavier workloads. We demonstrate that CFs are now able to run heavy workloads similar to VMs in our evaluation.

Using CFs with VMs: We create a system that allows sub-second launch of long-running processes. To achieve this, we implement the system shown in Figure 1a, consisting of cloud provided VMs and CFs plus a computer running our controller. AWS and Azure provide proprietary scaling of CFs; however, we control CF scaling directly to perform CF-to-VM hand-off. At startup, the controller creates a pool of CFs and loads applications on them. When the controller scales out, it determines the number of required VMs and the number of cores on each VM. It selects an idle CF from the pool and associates it with a core. The controller simultaneously starts the VM and instructs the CF to resume processing. The CF begins processing requests immediately and the VM continues to launch. When the VM is ready, the controller suspends the CF and returns it to the pool. When a CF is idle, the provider may swap it out of cache. Launching a CF swapped out of cache is a *cold start*; launching a cached CF results in a *warm start*. To minimize cold starts, the controller makes periodic requests to the CF [10]. This could be costly to the cloud provider if this technique were widely used, but each tenant needs very few warm CFs to mitigate VM launch latency. Regardless, our technique provides significant improvement since both warm and cold start CFs have significantly lower launch times than VMs. Cloud providers limit the amount of time a CF can process a single request. The maximum processing time can exceed the time it takes for a VM to launch. The controller continually makes sequential requests to the CF until its corresponding VM has launched, allowing the CF to run continuously until the VM is ready. AWS and Azure provide runtimes for languages including Java, JavaScript/NodeJS, and C#. It is possible to execute pre-compiled executables independent of the runtimes



(a) Scaling Architecture (b) Queuing Model

Fig. 1: Scaling System

on a CF [6], [20], [23] as described in §V. Cloud providers set limits on the number of VM cores a tenant may allocate simultaneously. We use a CF pool size equal to our core limit.

Model: We use the queuing model shown in Figure 1b which allows us to implement both publisher/subscriber (pub/sub) and request/response (req/resp) applications. It consists of client producers that generate jobs/requests, a queue to hold pending jobs/requests, and server consumers to process them. Our pub/sub implementation uses an explicit work queue to hold pending jobs. This implementation can be used for a wide range of applications such as back-end batch processors or IoT systems. Moreover, the queue need not be explicit. In our req/resp implementation, the queue consists of the TCP backlog and receive queues in a custom HTTP load balancer that we build. This implementation can be used for web or machine-to-machine (M2M) applications.

FEAT Scaling Algorithm: In state-of-the-industry scaling, tenants must specify rules on the provider's platform. Tenants define thresholds such as CPU utilization that trigger the rules. Tenants also specify how many cores to add or remove when the rule fires. Setting these values requires analyzing the system behavior in response to load patterns. When load patterns change, they may no longer be valid. We use *first principles* to develop a reactive algorithm that requires no pre-tuning, but quickly adapts to changing network conditions using runtime measurements. Our simple approach outperforms existing state-of-the-art approaches while eliminating the need to pre-specify thresholds which is a primary difficulty in other approaches.

Our algorithm (Algorithm 1) makes scaling decisions based on runtime measurements of the queue. It can execute at any frequency (§VII), but it is convenient to execute it in an event-driven manner whenever the cloud provider makes new measurements such as queuing delay available.

Scaling Out: Our scale out algorithm depends only on runtime measurements of arrival rate λ (offered load), queue growth rate g, and departure rate. Let c_0 be the number of cores before scaling and c_1 be the number after. The required number of cores to process the current load is λ/μ where μ is the processing rate per core. If a queue has formed before we can add cores, this ratio is insufficient to drain the queue. Our intuition is to include g to collapse the queue to zero delay over the next interval. The required cores c_1 is the arrival rate plus the queue growth rate divided by the processing rate per core as shown in Equation 1. When scaling horizontally and vertically, we give preference to launching VMs with the largest core count available, not exceeding those remaining to launch.

$$c_1 = \lceil \frac{\lambda + g}{\mu} \rceil \tag{1}$$

Scaling In: Scaling in is more difficult since we need to detect spare capacity in the system. Our intuition is that if the measured processing rate in two consecutive intervals is decreasing, less capacity is required and we scale in. We measure the number of jobs, n, processed in the first interval and the time it takes to process the jobs, $\sum_{i=1}^{n} J_i$ where J_i is the time it takes to process job i. The number of opportunities available to process jobs is $\alpha_0 = (n/\sum_{i=1}^{n} J_i)t$ where t is the time since the last iteration. We then measure the number of jobs α_1 processed in the second interval. The ratio $h = \alpha_1/\alpha_0$ gives the required decrease in capacity. The new number of cores is given by Equation 2.

$$c_1 = \lceil \frac{\alpha_1}{\alpha_0} c_0 \rceil \tag{2}$$

Combined Scaling: If the queuing delay increases, we scale out according to Equation 1. If the queuing delay does not increase, we scale in according to Equation 2. When scaling out, we verify the required capacity is greater than the predicted capacity. We do this to check our measurements which can be in error because of timing or high variability in offered load. If we detect an error ($k < h$), we do not scale, but collect more measurements over the next interval. When scaling

Algorithm 1 FEAT Scaling Algorithm

```
c0 = current number of cores
c1 = new number of cores
t = time since last iteration
d = measured change in queuing delay
g = measured queue growth rate in jobs/sec
λ = measured job arrivals/sec
x = measured job departures/sec
μ = x/c0 = departure rate per core
k = (λ + g)/μ = required capacity
α1 = jobs processed in current iteration
n = jobs processed in previous iteration
Ji = time to process job i in previous iteration
α0 = nt/(Σ_{i=1}^{n} Ji) = predicted opportunities
h = α1/α0 = predicted capacity
if (d > 0) { // queue is growing, scale out
 if (k < h) {
  // Measured capacity < predicted but
  // queue is growing. Invalid
  // measurement or change in traffic.
  // Wait for more measurements.
  c1 = c0
 } else { c1 = ⌈(1/μ)(λ + g)⌉) }   // Equation 1
} else { // queue is not growing, scale in
 if (1 < h) {
  // Queue is not growing but prediction
  // indicates scale out. Wait for more
  // measurements, do not scale.
  c1 = c0
 } else { c1 = ⌈(α1/α0) c0⌉ } }      // Equation 2
return (c1)
```

in, our prediction can be in error when load patterns change. Before scaling in, we verify the ratio $h = \alpha_1/\alpha_0$ is less than 1, indicating we have excess processing opportunities.

IV. Cost Analysis

FEAT saves costs in two ways from a tenant's perspective. One depends on the provider's CF and VM cost structure. The other is in terms of latency violations.

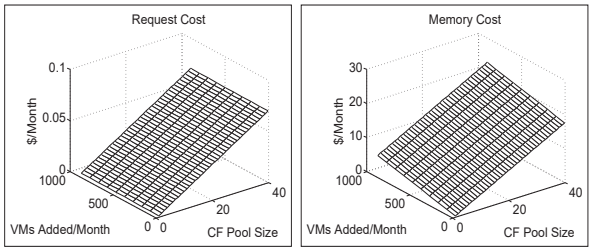
Provider-Dependent: We acknowledge that the costs in this subsection will vary as providers change their pricing model. However, it is likely that CFs will continue to be less expensive over short durations than VMs for both the provider and tenant because CFs are implemented with containers that have less fragmentation than VMs and can share physical resources more efficiently than a VM. Cloud providers price CFs differently than VMs. VMs are intended to run for long periods of time and AWS and Azure bill by execution time. CFs are intended to handle a large volume of requests and AWS and Azure bill in terms of number of req/month and the amount of memory used during each request. FEAT incurs cost only while launching a VM or keeping a CF warm. It uses few req/month so memory is the dominant cost. Our system keeps a CF warmed by making short running requests every 5 minutes. These short running requests last less than 1 second in duration, but we round up to 1 second here to simplify the analysis.

Let C be the number of CFs in the pool. Let V be the number of VMs added/month. Let A be the cost/req and B be the cost per GB-sec. Total cost per month is $(C_{req}+V_{req})A+(C_{mem}+V_{mem})B$ where C_{req} is the number of req/month to keep the CF warm, V_{req} is the number of req/month to launch a VM, C_{mem} is the amount of memory/month to keep the CF warm, and V_{mem} is the amount of memory/month to launch a VM. While a VM is initializing, we make requests at a rate of 1 req/min (the provider's measurement frequency) to keep the application executing on the CF. When the VM is running, we allow the CF to return to an idle but warmed state. At the time of this writing, $A = \$2 \times 10^{-7}$ and $B = \$1.667 \times 10^{-5}$ for both AWS and Azure. If it takes 2 min to launch a VM and 3 GB RAM/CF (the worst case on AWS), then we have $C_{req} = 8,760\ C$, $V_{req} = 2\ V$, $C_{mem} = 26,280\ C$, and $V_{mem} = 360\ V$.

Figure 2 shows the cost for a range of CFs and VMs. Concretely, the cost of running a spare idle VM with 1 CPU and 3 GB RAM for a month is about \$50; whereas, the cost of keeping a similarly provisioned CF warm for a month is about \$0.50. It is two orders of magnitude less expensive for a tenant to keep a CF ready to process requests than an idle VM.

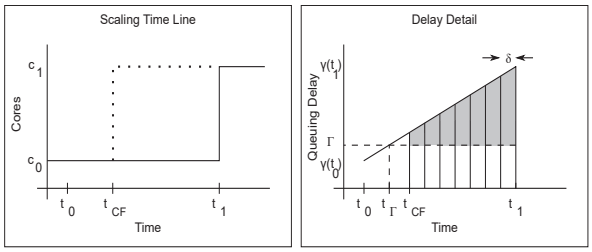
Provider-Independent: We now present our model for the cost savings during a scale out event. This model is independent of the cost structure of the provider. For this model, we assume latency is caused by queuing delay. Let Γ be a target latency. If mean queuing delay γ is greater than Γ during a time interval δ, we apply a cost penalty of β to the tenant's utility U [24]. The cost P of the request due to the penalty is $P = \beta\frac{\gamma}{\Gamma}U$.

Consider an offered load increase from λ_0 to λ_1 jobs/sec. Figure 3a shows the time line of a scale out event. At time t_0, the system is running with c_0 VMs and the controller makes



(a) Request Cost (\$) (b) Memory Cost (\$)

Fig. 2: Cost



(a) Time Line (b) Delay Detail

Fig. 3: Scale Out Event

a scaling decision. The dotted line shows the time line if the system uses CFs and the solid line without them. Time t_{CF} is the time at which the CFs become ready if the system is using them. Latency stops growing at t_{CF} if CFs are being used. Time t_1 is the time at which the newly launched VMs become ready. The queuing delay increases during the time from t_0 until the CFs or VMs become ready. Figure 3b shows the time series of the increase in delay during the scaling event. While the expected value of queuing delay grows exponentially with utilization, the time series growth is linear [8]. At some time t_Γ, the queuing delay becomes greater than the target Γ. Latency violations occur in the shaded area of the figure. The total penalty is the sum of the penalties in each δ.

We use constant values of λ_0 and λ_1 in the figures to illustrate our point; however, in a real system, the arrival rate will vary over time. To compute the penalty at runtime, a real system cannot simply compute the area of the shaded portion of Figure 3b. We compute the total penalty P_{total} at runtime by summing the penalties at the end of each interval δ. There are $\frac{t_1-t_\Gamma}{\delta}$ intervals between t_Γ and t_1. Let γ_i be the measured average latency between times $t_\Gamma + (i-1)\delta$ and $t_\Gamma + i\delta$ where i is the interval number, then we have $P_{total} = \sum_{i=1}^{\frac{t_1-t_\Gamma}{\delta}} \beta\frac{\gamma_i}{\Gamma}U$.

Figure 4 shows a sample cost savings as a function of the increase in offered load. Let P_{CF} be the penalty when using CFs and P_{VM} be the penalty when not using them. The cost savings is $\Delta P = P_{VM} - P_{CF}$. For this example, we use values on on the order of those in our evaluation.

Key observations: There is a fundamental cost advantage to decreasing the effective launch time of a VM. As a continuous function, the savings is an integral of a linear equation

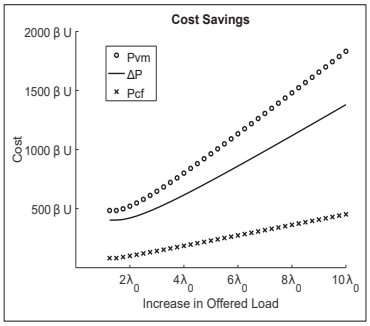


Fig. 4: Cost Savings During Scale Out Event

describing the increase in latency in Figure 3b. This manifests as a parabola in Figure 4. Consequently, there is higher cost savings for larger increases in offered load.

V. Implementation

We implement FEAT and three other approaches for comparison: (1) a constant number of cores (CONST), (2) a pre-tuned CPU utilization-based algorithm (MAN), and (3) a state-of-the-art hybrid algorithm [24] (FINE). CONST represents a statically-provisioned production system from which we obtain network traces in our evaluation. MAN is a manually pre-tuned state-of-the-industry algorithm available from most cloud providers. FINE is an existing state-of-the-art algorithm using a hybrid of predictive, reactive, and feedback control techniques. We implement a scalable pub/sub and a load balanced req/resp web server farm. We write custom producer, consumer, and controller applications in Java except for the Azure CF shell which we write in C#. The two systems are similar in structure with details below. On AWS, we use EC2 (for VMs) and Lambda (AWS's CFs). On Azure, we use the Virtual Machines and Functions offerings. The producer reads and recreates a pre-recorded network trace. It creates jobs (for pub/sub) or requests (for req/resp) and sends them to the job queue or load balancer.

Publisher/Subscriber: We use Apache ActiveMQ [3] for our job queue. The producers (publishers) send the job data to the work queue. The consumers (subscribers) pull jobs from the queue on a first-come-first-served basis. A consumer uses CPU resources proportional to the job size.

Web Server: For this application, the producers are HTTP clients. They make requests to a custom load balancer that we write. The load balancer proxies requests to Apache HTTP [4] running on the VMs and CFs. Apache HTTP proxies the request to the consumer on the same VM or CF. The consumer returns the response which follows the reverse path back to the client. The queue consists of the TCP backlog and receive queues in the load balancer. We set the backlog and timeouts high enough that no requests are dropped to avoid reducing to tail drop.

AWS Lambdas are preloaded with Java, JavaScript/NodeJS, Python, and C# runtimes on an Amazon Linux OS. Azure Functions are preloaded with C#, F#, JavaScript/NodeJS, Python and PHP runtimes on the Windows Server 2012 Datacenter OS. Programmers write code for any of these runtimes with an SDK from the provider and upload it to the provider's CF system. A CF will run any code that compiles to a user-level executable [6], [23]. We regard our CF as a shell that interfaces to the underlying OS. We execute and control our consumer and Apache HTTP which do not depend on the supported runtimes. We write our shells in Java on AWS and in C# on Azure. We write our consumer in Java. Apache HTTP is written in C.

The CFs resume from a paused state in 20-40 sec on AWS and 40-50 sec on Azure. The VMs launch and start our applications on AWS in 2-3 min and 4-5 min on Azure. Azure queuing delays grow more than AWS before VMs are fully launched. As a result, Azure benefits more from CFs than AWS.

VI. Evaluation

We first evaluate the algorithms with a synthetic data set without CF fast launch to establish a baseline. Next, we add fast launch to show the improvement from decrease in effective launch times. Finally, we evaluate them on a network trace from a statically and over-provisioned production web server to determine how robust they are in a real-world environment. We summarize mean results in Tables I and II. Note that the results for all algorithms show an average queuing delay an order of magnitude larger than typical service level agreements (SLAs) [5]. In a real-world web server, when the TCP request rate exceeds capacity, the backlog fills and the server drops arriving TCP connections. The queuing delay will not grow very large. In our pub/sub application, we use an explicit queue to hold pending jobs and in our web server application, we set the TCP backlog to a very high value to evaluate the system without resorting to tail drop. We do not discard any requests. Consequently, queuing delay can grow large and stay at elevated levels until we add capacity to the system.

CONST, MAN, and FINE require pre-tuning to the network. Fast launch has no significant effect on CONST because its VMs are always running. FEAT does not require pre-tuning.

FINE: FINE trains an autoregressive AR(2) predictor ahead of time on pre-recorded traces to predict arrival rate [24]. It pre-computes a lookup table that maps arrival rate to the core count required to service that rate. At runtime, it looks up the predicted rate to obtain the core count. It uses latency as a feedback signal to fine-tune the core count to achieve a target latency. One issue we encounter while pre-tuning FINE is that pre-computing the core count does not always give an appropriate computing capacity because workloads vary over time. The feedback signal fine-tunes the capacity but introduces feedback delay. Another issue is pre-tuning the feedback gain. At high gain, the core count far exceeds a provider-imposed core quota. In spite of our best effort, the core count occasionally exceeds the quota during the evaluations. We limit the core count to the quota in this case. We use the algorithm's default 0.6 ms target latency and a gain of 10^{-5}.

MAN: If the average CPU utilization exceeds an upper threshold, MAN scales out by a pre-tuned number of cores. If it drops below a lower threshold, MAN scales in by another pre-tuned number of cores. We find that an upper threshold of 40% and a lower threshold of 20% work well in our environment.

TABLE I: Synthetic Summary. Pub/sub is listed on the left side of each column, req/resp on the right.

Experiment	Scaling	Algorightm	Queuing Delay (sec)				Number of Cores				CPU (%)			
			AWS		Azure		AWS		Azure		AWS		Azure	
Square Wave without CF Fast Launch	Horizontal Only	CONST	24.02	25.59	34.38	22.40	12.00	12.00	12.00	12.00	24.65	24.54	24.64	22.79
		MAN	217.55	224.20	285.15	292.55	9.48	9.41	9.00	8.57	45.66	43.76	40.41	42.99
		FINE	50.65	51.81	101.47	133.15	8.56	10.13	11.51	13.14	57.29	56.63	46.21	46.59
		FEAT	45.54	31.31	83.61	101.59	7.35	7.50	8.98	9.54	45.77	45.04	41.98	40.94
	Horizontal and Vertical	CONST	23.01	24.51	25.52	25.25	12.00	12.00	12.00	12.00	30.19	29.15	29.26	33.26
		MAN	232.62	242.65	388.29	318.19	8.93	9.34	7.30	7.91	43.72	43.33	44.43	44.30
		FINE	53.40	60.43	88.87	116.98	9.09	9.98	9.69	13.29	51.30	48.13	46.22	43.16
		FEAT	43.20	44.28	70.74	95.28	8.33	7.13	11.42	11.46	43.10	43.26	34.76	39.99
Square Wave with CF Fast Launch	Horizontal Only	CONST	23.90	25.62	23.69	21.98	12.00	12.00	12.00	12.00	24.25	23.71	26.43	21.71
		MAN	148.05	169.16	247.20	203.59	7.62	7.70	7.52	7.27	43.54	42.64	42.14	38.74
		FINE	41.28	31.31	41.73	34.79	9.09	8.62	12.26	10.41	56.65	51.05	40.87	40.94
		FEAT	25.71	18.44	30.41	19.77	6.48	6.61	8.92	8.93	45.41	38.52	37.14	35.53
	Horizontal and Vertical	CONST	23.01	24.66	24.73	22.16	12.00	12.00	12.00	12.00	29.59	28.91	27.04	25.78
		MAN	202.07	114.90	222.67	230.61	9.02	7.96	7.33	7.27	41.19	37.34	40.39	40.04
		FINE	33.85	28.17	34.28	47.58	8.29	7.91	10.28	14.27	54.85	49.39	42.25	40.28
		FEAT	24.66	25.69	20.76	51.69	7.96	8.89	10.37	9.73	42.88	37.44	36.44	36.83

We set the upper threshold somewhat low to detect increased load as early as possible. We scale by ± 3 cores per operation.

CONST: We set the number of cores in each experiment in CONST to handle 3/4 of the peak request rate. The choice of core count is a trade-off between the cost of over-provisioning and the cost of slow response at high workloads. At low request rates, CONST is over-provisioned and wastes resources. At high request rates, it is under-provisioned and incurs long queuing delays. Provisioning CONST is difficult because it may be hard or even impossible to foresee peak rates in real-world systems.

Observations: In general, without fast launch, we observe a lower queuing delay with horizontal over combined horizontal and vertical scaling. This is caused by variance in the time it takes to launch a VM. With more smaller VMs, some launch quickly and begin to process requests while the remaining VMs continue to launch. Adding CFs tends to equalize this difference. We see lower queuing delay in AWS than Azure without fast launch because of the longer VM launch times on Azure. Our use of CFs tend to decrease this difference.

A. Experiments

Without Fast Launch: We establish a baseline for each algorithm by disabling the CFs. We generate a square wave with a trough of 15 req/sec and two peaks of 35 req/sec (more than $2\times$ peak-to-trough) and 150 req/sec ($10\times$ peak-to-trough). We alternate peak rates every period. We set the peak and trough durations to 10 min each. We evaluate first with horizontal scaling only and then with both horizontal and vertical scaling. With horizontal only, we use VMs with a single core.

We set CONST to 12 cores. It has lowest queuing delay because it is over-provisioned most of the time. This is reflected in its low CPU utilization.

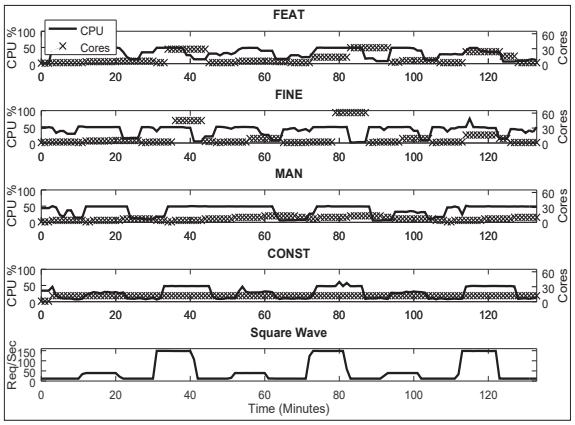
With Fast Launch: We see a significant improvement in queuing delay with CFs enabled though they have no real effect on CONST since its cores are always running. Our CFs allow capacity to be quickly added and higher request rates have less opportunity to form a queue. We see up to 45% less queuing delay on AWS and up to 75% on Azure. We see up to 20% reduction in cores in most cases. The representative traces in Figure 5 show the combined horizontal and vertical scaling results for pub/sub on Azure. We see FINE compute a core count beyond our quota near the 80 minute mark. We see the queuing delay drop, but CPU utilization also drops, causing inefficient use of resources for an extended period. CONST is not able to control queuing delay during the peak traffic rates.

Production Trace Evaluation: We collect network traces from a production system that serves assets and metadata to hundreds of clients. It is over-provisioned with multiple machines running Apache HTTP. It services both M2M and web browser requests. We execute many traces with similar results. To illustrate the issues with sudden changes in offered load, we choose a trace from a 24-hour period in February 2017 with a sudden spike in traffic near the 17th hour shown in Figure 6. The figure shows req/sec, response size, and response size distribution. We execute our web server with CF fast launch enabled and horizontal scaling. We show AWS results in Figure 7. We break queuing delay results into 1-min intervals and show the percentage of delays greater than 10 ms and 50 ms in these intervals, representing percentage of SLA violations. In this experiment, we run CONST over-provisioned (CONST1) and again with enough cores to handle $3/4$ peak rate (CONST2). We show the latter run in the figure.

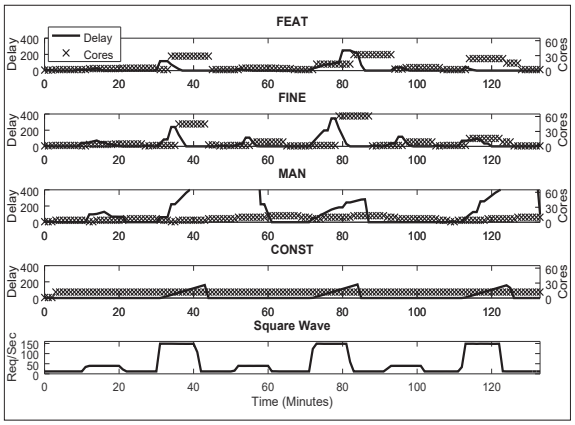
CONST1 data shows inefficient use of cores (45% CPU utilization on AWS and 42% on Azure). It has fewer SLA violations, but at higher core count. CONST2 queuing delay grows to excessive levels and the system does not recover by the end of the experiment. FINE performs well because there is less variability in request rate than in our square wave evaluations; however, FEAT performs nearly equally well without the need to pre-tune knobs. AWS results show FEAT decreases queuing delay by 40% over MAN and improves CPU utilization by 40% compared to over-provisioned CONST1. This indicates more efficient use of resources, reflected in the decrease in core count. FEAT decreases the core count, and thus cost, by

TABLE II: Production Summary, Request/Response

Algorithm	Queuing Delay (sec)		Number of Cores		CPU (%)		10 ms Violations (%)		50 ms Violations (%)	
	AWS	Azure	AWS	Azure	AWS	Azure	AWS	Azure	AWS	Azure
CONST1	0.01	0.01	5.00	5.00	45.08	42.00	3.55	2.15	2.86	1.32
CONST2	662.99	242.68	3.00	3.00	73.42	75.03	36.49	34.67	35.52	33.22
MAN	1.05	0.47	4.07	3.42	62.97	69.20	14.93	15.77	13.43	14.51
FINE	0.73	0.32	3.14	3.21	82.04	80.62	10.67	6.74	10.54	5.37
FEAT	0.66	0.24	3.19	3.20	79.16	77.92	10.57	6.53	10.44	5.21



(a) CPU % and Cores



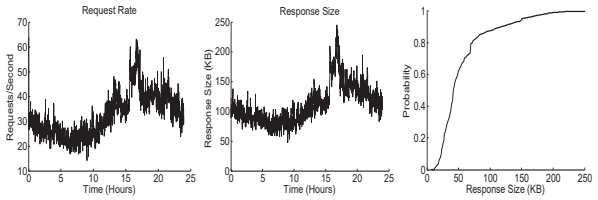
(b) Queuing Delay (Sec) and Cores

Fig. 5: Azure Pub/Sub with Fast Launch

about 20% compared to MAN and 35% compared to CONST1.

B. Measurement Periods

Cloud providers periodically measure VM CPU utilization and managed queue delays, eliminating the need to modify existing code to obtain these measurements. AWS and Azure offer two periods, a 5 min free tier and a 1 min paid tier. Since FEAT is event-driven and operates at the cloud provider's measurement period, we evaluate it at each. We run a square wave with fast launch enabled. We set a 35 req/sec peak and 15 req/sec trough with durations of 30 min each and scale horizontally. Table III shows the results. FEAT performs much better at 1 min intervals by keeping delays very low with little change in the number of cores and CPU utilization. Therefore,



(a) Request Rate (Requests/Sec) (b) Response Size (KB) (c) Response Size Distribution

Fig. 6: Network Trace

TABLE III: Measurement Period Summary

Period	Delay (Sec)		No. Cores		CPU (%)	
	AWS	Azure	AWS	Azure	AWS	Azure
1 min	0.47	0.33	2.97	2.82	70.35	69.24
5 min	8.71	7.89	2.79	2.76	72.52	74.32

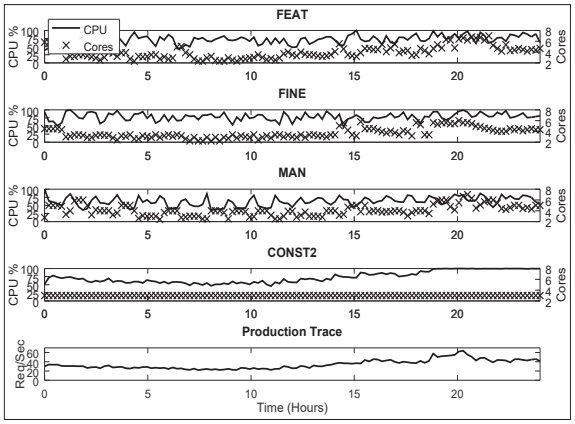
we use a 1 min interval for all experiments in §VI. The best measurement period among those offered by the cloud provider can be determined dynamically at run-time as a background task. We omit the details of this task due to space limitations.

VII. Stability

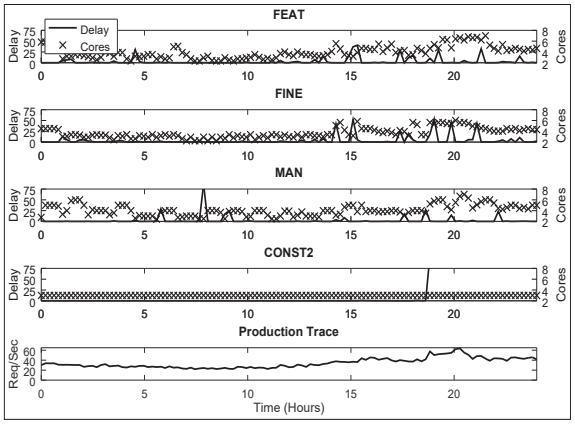
We evaluate FEAT's stability with a discrete-event simulator, that we build, that models the system as an M/M/c queue [8] with c cores. Although M/M/c is not necessarily the perfect model of FEAT, it gives a basis to evaluate stability. Arrival and processing rates have exponential distributions with means λ and μ, respectively. We run the simulator on a wide range of values to probe boundaries of stability. Experiments include (1) a single arrival rate with mean λ, (2) a square wave with λ alternating between two values, and (3) a wave where we vary λ as a sinusoid. We measure mean queuing delay and compute standard deviation and inter quartile range (IQR) of the queuing delay. We measure the mean core count and utilization. Table V shows representative results for experiment 1 with $\lambda = 15$ jobs/sec, $\mu = 10$ jobs/sec, and launch time $= 60$ sec. IQR remains low and utilization indicates the queuing system remains stable for all cases in all experiments. We observe the same general trends in simulation as in the implementations. With shorter metric period, queuing delay decreases, core count increases, and CPU utilization decreases.

VIII. Conclusion

We built a novel architecture for quickly auto-scaling virtual machines by using *cloud functions* and designing an algorithm



(a) CPU% and Cores



(b) Queuing Delay (Sec) and Cores

Fig. 7: AWS Req/Resp Production Trace

TABLE IV: Simulation Ranges

Variable	Minimum	Maximum
Arrival Rate λ	1 job/sec	1000 job/sec
Processing Rate μ	1 job/sec	1000 jobs/sec
Launch Time (Sec)	0 sec	10 min
Measurement Period	10 sec	10 min

that does not require pre-specification of parameters or thresholds. We implemented our system on both AWS and Azure. We compared it with pre-tuned and statically-provisioned algorithms as well as a state-of-the-art hybrid algorithm. We evaluated it on both synthetic data sets and production network traces for both horizontal only and combined horizontal and vertical scaling. We determined that it is stable through extensive simulation. We found that our system can reduce latency by $2\times$ and reduce the number of VMs, and thus cost, by 20%.

TABLE V: Results for Simulation Experiment 1

Period (s)	Delay (ms)	IQR (ms)	Std Dev (ms)	CPU (%)	No. Cores
10	51.63	0.05	85.47	54.56	2.87
100	98.85	0.10	157.40	67.06	2.33
1000	124.43	0.19	203.48	71.28	2.16

ACKNOWLEDGMENTS

This material is based upon work supported by the National Science Foundation under Grant No. 1302688.

REFERENCES

[1] Amazon, "Amazon web services lambda," http://aws.amazon.com/lambda, 2017, accessed: 2017-06-17.

[2] ——, "Aws lambda doubles maximum memory capacity for lambda functions," http://aws.amazon.com/about-aws/whats-new/2017/11/aws-lambda-doubles-maximum-memory-capacity-for-lambda-functions, 2017, accessed: 2018-02-03.

[3] Apache, "Apache activemq," http://activemq.apache.org, 2017, accessed: 2017-06-17.

[4] ——, "Apache http server project," http://httpd.apache.org, 2018, accessed: 2018-01-29.

[5] G. DeCandia *et al.*, "Dynamo: amazon's highly available key-value store," in *ACM SIGOPS op sys review*, vol. 41. ACM, 2007, pp. 205–220.

[6] S. Fouladi *et al.*, "Encoding, fast and slow: Low-latency video processing using thousands of tiny threads." in *NSDI*, 2017, pp. 363–376.

[7] A. Gandhi *et al.*, "Autoscale: Dynamic, robust capacity management for multi-tier data centers," *ACM TOCS*, vol. 30, no. 4, p. 14, 2012.

[8] N. Gautam, *Analysis of queues: methods and applications*. CRC Press, 2012.

[9] S. Hendrickson *et al.*, "Serverless computation with openlambda," in *8th USENIX Workshop on Hot Topics in Cloud Computing, HotCloud 2016, Denver, CO, USA, June 20-21, 2016.*, 2016. [Online]. Available: https://www.usenix.org/conference/hotcloud16/workshop-program/presentation/hendrickson

[10] A. Hornsby and N. Undén, "Getting started with aws lambda and the serverless cloud," https://www.slideshare.net/AmazonWebServices/aws-lambda-and-serverless-cloud-61712836, 2016, accessed: 2017-09-11.

[11] IBM, "Ibm bluemix openwhisk," http://www.ibm.com/cloud-computing/bluemix/openwhisk, 2017, accessed: 2017-06-17.

[12] J. Jiang *et al.*, "Optimal cloud resource auto-scaling for web applications," in *Cluster, Cloud and Grid Computing (CCGrid), 2013 13th IEEE/ACM International Symposium on*. IEEE, 2013, pp. 58–65.

[13] E. Jonas *et al.*, "Occupy the cloud: distributed computing for the 99%," in *2017 Symposium on Cloud Computing*. ACM, 2017, pp. 445–451.

[14] H. Lim *et al.*, "Automated control in cloud computing: challenges and opportunities," in *Proc of the 1st workshop on Automated control for datacenters and clouds*. ACM, 2009, pp. 13–18.

[15] ——, "Automated control for elastic storage," in *Proc of the 7th international conf on Autonomic computing*. ACM, 2010, pp. 1–10.

[16] M. Mao and M. Humphrey, "A performance study on the vm startup time in the cloud," in *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*. IEEE, 2012, pp. 423–430.

[17] Microsoft, "Microsoft azure functions," http://azure.microsoft.com/en-us/services/functions, 2017, accessed: 2017-06-17.

[18] P. Nasirifard, "A serverless topic-based and content-based pub/sub broker," in *Proceedings of the 18th ACM/IFIP/USENIX Middleware Conference: Posters and Demos*. ACM, 2017, pp. 23–24.

[19] H. Nguyen *et al.*, "Agile: Elastic distributed resource scaling for infrastructure-as-a-service." in *ICAC*, vol. 13, 2013, pp. 69–82.

[20] A. Singhvi *et al.*, "Granular computing and network intensive applications: Friends or foes?" in *Proceedings of the 16th ACM Workshop on Hot Topics in Networks*. ACM, 2017, pp. 157–163.

[21] G. Tesauro *et al.*, "A hybrid reinforcement learning approach to autonomic resource allocation," in *International conf on Autonomic Computing (ICAC'06)*. IEEE, 2006, pp. 65–73.

[22] B. Urgaonkar *et al.*, "Dynamic provisioning of multi-tier internet applications," in *Second International Conference on Autonomic Computing (ICAC'05)*. IEEE, 2005, pp. 217–228.

[23] T. Wagner, "Running arbitrary executables in aws lambda," https://aws.amazon.com/blogs/compute/running-executables-in-aws-lambda, 2015, accessed: 2017-3-15.

[24] C. Wang, A. Gupta, and B. Urgaonkar, "Fine-grained resource scaling in a public cloud: A tenant's perspective," in *Cloud Computing (CLOUD), 2016 IEEE 9th International Conference on*. IEEE, 2016, pp. 124–131.