Happiness index: Right-sizing the cloud's tenant-provider interface

Vojislav Dukic, Ankit Singla
Department of Computer Science, ETH Zurich

Abstract

Cloud providers and their tenants have a mutual interest in identifying optimal configurations in which to run tenant jobs, *i.e.,* ones that achieve tenants' performance goals at minimum cost; or ones that maximize performance within a specified budget. However, different tenants may have different performance goals that are opaque to the provider. A consequence of this opacity is that providers today typically offer fixed bundles of cloud resources, which tenants must themselves explore and choose from. This is burdensome for tenants and can lead to choices that are sub-optimal for both parties.

We thus explore a simple, minimal interface, which lets tenants communicate their happiness with cloud infrastructure to the provider, and enables the provider to explore resource configurations that maximize this happiness. Our early results indicate that this interface could strike a good balance between enabling efficient discovery of application resource needs and the complexity of communicating a full description of tenant utility from different configurations to the provider.

1 Introduction

Ideally, cloud providers would provision for each tenant exactly the cloud resources that strike the tenant's desired cost-performance targets. This would fulfill both the tenants' and the providers' goals: each tenant pays for exactly what they get, and the provider uses their infrastructure at its maximum efficiency, without uncertainty in how to provision resources for tenants. However, there is no effective mechanism today for tenants to understand and convey their precise needs to the providers.

Cloud data centers allow tenants to run applications in a variety of configurations, using cloud virtual machines or instances that differ in available memory, CPU cores and clock speeds, and storage and networking options. They also expose several higher-level services such as relational and non-relational databases, and machine learning and stream processing engines, which often accommodate their own tenant-specific configurations. A tenant's choice of configuration can have a large impact on their cost and the performance achieved for their workload.

Finding appropriate configurations that achieve the desired cost and performance targets is a non-trivial exercise, and in the face of a large number of available configuration options, burdensome for cloud tenants. Even just launching a virtual machine in today's public cloud platforms requires choosing from among tens of possible VM configurations [1, 3, 4].

This is a well-studied problem, and there is already substantial work on identifying suitable configurations out of a given set, *e.g.,* identifying out of the tens of bundled virtual machine types on offer, the right one to use for a particular workload [5, 13, 15, 17]. However, these efforts still place the burden of exploring configurations on tenants. Additionally, the configuration chosen by the user might not be optimal from the cloud's standpoint, *e.g.,* a particular machine type may be in high demand, while others with similar performance for many of those workloads remain idle.

Leaving the exploration of appropriate configurations to tenants (regardless of whether from a sparse space of bundled configurations, or a more continuous spectrum) has three disadvantages: (a) decreased usability and a higher barrier to entry for new tenants; (b) tenants would incur additional expenses for such exploration, which the provider could avoid or reduce by conducting the exploration at times of low utilization; and (c) with dynamic pricing, the desirable configuration changes with time, requiring continuous vigilance and effort by the tenant. Thus, an approach where the provider can auto-configure tenants' execution environments would be invaluable in improving usability and efficiency. A key roadblock to the provider themselves searching for optimal configurations is the lack of information: they do not have any visibility into a tenant application's performance goals, so how should they optimize for the tenant?

There is a range of possibilities to bridge this tenant-provider divide, defined by the amount of information exchanged between the two. A significant body of past work has explored the extremes: (a) the tenant provides zero information, such that any optimization run by the cloud provider is based on monitoring metrics like CPU or memory utilization [11, 12, 15]. However, such metrics may not have a simple, clear relationship to application-level objectives; and (b) the tenant describes in detail their application's utility across

fine-grained collections of available resources [8, 16]. These extremes illustrate the tradeoff: the former limits optimization, and hence efficiency, at the expense of a simpler interface and usability, while the latter admits precise optimization, but requires substantial effort from the tenant, together with a complex tenant-provider interface capturing general utility functions across potentially many types of resources.

The clear deficiencies of the above two extreme variants of the tenant-provider interface motivate the central question of this work: *does an appropriate interface exists which balances the goals of efficiency and usability?* Our early results on exploring various design options for this interface reveal that a simple-to-use interface could potentially also achieve high efficiency. This interface consists of the tenant reporting to the provider their **happiness index**, a real value $\in [0, 1]$ representing the tenant's satisfaction with the current performance. For simple scenarios, which may themselves cover several use cases, we show, somewhat surprisingly, that this minimal interface suffices for the provider to rapidly discover cheap and efficient configurations.

Our proposed tenant-provider interface, based on the happiness index, offers both simplicity and efficiency and could lead to a substantial change in how cloud services work. However, more complex scenarios pose several challenges to our approach, including the expansion of the search space of configurations. We thus also elucidate such issues, and potential approaches for addressing them, in the hope that this work ignites a broader debate about what the right interface is, and the tradeoffs involved therein.

2 A Happy Cloud

We explore a simple interface between the cloud provider and tenant, whereby the tenant communicates to the provider a single, real-valued evaluation of their satisfaction from the current configuration, and the provider uses the evaluation to adjust the resource configuration, hopefully converging towards high-efficiency configurations. We first provide a simplified description, leaving several practical issues to §6.

2.1 What does the tenant provide?

The tenant must translate the metrics they care about, such as Web server response time, rate of tuples processed in a stream processor, job completion time, etc. to the happiness index, a real value $\in [0, 1]$, with 1 being the maximum possible satisfaction with the service. Note that the tenant's calculation can combine arbitrary metrics (averages, percentiles, etc.) and be arbitrarily complex, but is opaque to the provider, who only sees the happiness index. This is key to eliminating the need for the provider to understand in detail the metrics and structure of the tenant's application.

It must also be clear what resources the reported happiness applies to, *i.e.,* the "happiness domain". This could be any combination of virtual machines, lambda functions, or cloud-offered services. By default, it is assumed that all services used by the tenant are part of this happiness domain, and no additional input is required. However, if the tenant is running multiple independent high-level applications, these must be identified as separate happiness domains, with their happiness index values reported separately.

Implementation: Should the provider query the tenant for their happiness? This could simplify optimization at the cloud provider, *e.g.,* the optimizer could decide when new data is useful, and adjust the rate of these queries depending on convergence rate and the behavior of the application. However, this limits flexibility for tenants: they must make happiness values available on demand, potentially restricting their options for implementing their calculation. Additionally, for some applications, it may only be possible to provide metrics after completion. Thus, we suggest that tenants post happiness values as they compute them to an agreed-upon location, *e.g.,* an HTTPS URL.

2.2 What does the provider do?

The provider receives the happiness index values the tenant reports and uses them to decide whether or not to test new resource configurations, and if so, which ones. To accommodate tenant expense budgets and a notion of service-level agreement, the provider can accept from a tenant either of cost or happiness thresholds as a constraint, and optimize for the other, *i.e.,* maximize the tenant's happiness within a fixed budget, or minimize their cost while guaranteeing at least a certain target happiness. The provider's objective is to use reported happiness values to optimize the configuration.

Optimization dimensions: Every happiness domain has a set of parameters that can be optimized, including hardware parameters like bandwidth, memory, the number of CPU cores, the type of storage; and software parameters for cloud services that the tenant uses, such as cloud-managed databases. Beyond these configuration options that tenants manually configure today (at a coarse granularity), providers may also explore parameters that likely cannot be exposed to tenants, such as packet priorities. The parameters explored may depend on the happiness domain: the parameters for a domain containing only a single virtual machine may be very different from that for a complex service with different components running on several virtual machines.

Implementation: The provider must use a sparse sampling of the utility of a small number of configurations to search for near-optimal configurations. This implies black-box optimization, in the absence of a "gradient" that could be used in methods like stochastic gradient descent. Further, the samples of utility in the form of happiness index can be noisy, as even for simple applications, there can be substantial variation in performance and application load over time.

While our interface could be implemented using a variety of methods, such problems are often amenable to Bayesian optimization [9, 14], which allows gradient-free optimization by modeling the objective as a Gaussian process. Based on

available samples of the objective function, it estimates the potential improvement in the objective value from every possible set of inputs, allowing an efficient search of the input space. While this type of optimization has been used in other systems (such as those for helping tenants explore configurations [5]), it can violate tenant-specified bounds on cost or happiness during its search procedure.

To guarantee adherence to tenant cost or happiness constraints, we use a *safe* variant of Bayesian optimization [6, 7], whereby the parameter exploration is performed in such a way that it never violates a specified constraint. This method has been applied to improve the controllers of physical systems like robots and quadrotors. The safety criteria slow down convergence but admit SLA enforcement.

Caveat: Any safe optimization approach requires a safe starting point, *i.e.,* the tenant must provide an initial configuration that meets their specified bound on cost or happiness. In the absence of a safe starting point, no safe exploration is possible. However, we believe this is not a difficult requirement to fulfill in practice: tenants typically overprovision to meet their performance targets [10], and such configurations would provide a good starting point for optimization towards cheaper configurations. Relatedly, this approach can only search through a convex space of configurations which includes the starting safe configuration.

3 Preliminary results

As a proof-of-concept for our approach, we test a simple happiness domain containing one virtual machine and a database service and optimize three types of resources:

- Available network bandwidth: using Linux `tc`, we vary the bandwidth between 10 Mbps and 1 Gbps.
- CPU cycles: we vary maximum CPU utilization between 20% and 100%.
- Database: we use Azure CosmosDB, varying the maximum request units from 400 to 5000.

We describe resource utilization in terms of percentage of the available range, *e.g.,* using 2300 DB request units translates to a $\frac{2300}{5000-400} = 0.5$ usage of DB resources. The cost of an individual resource is also simply its fractional utilization (0.5 in this case); with the cost of a set of resources being the average of different resource utilization. These choices are largely arbitrary, and in our experience, do not affect our results substantially.

To explore scenarios that use different combinations of resources, we set up a Web server that can respond to 3 distinct types of queries, with each query type stressing one of the above 3 resources. Bandwidth-focused queries fetch static Web pages from the server, uniformly distributed in the 1-5 MB range; CPU-focused queries force the server to calculate the first N prime numbers for $N \in [80k, 120k]$; and the database-focused queries result in the server adding, updating,

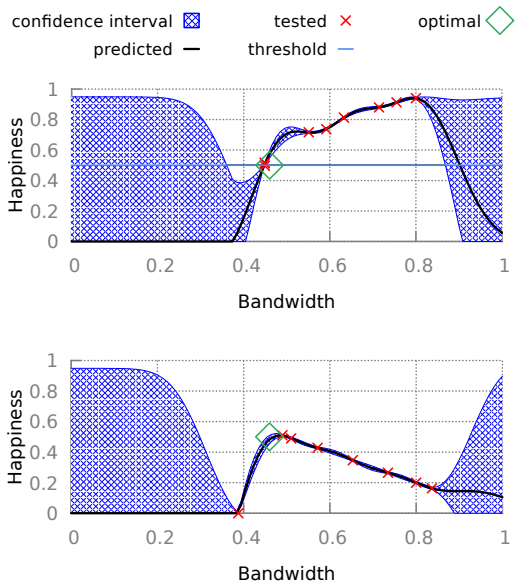

Figure 1: *The optimization process for the bandwidth-focused experiment. The optimizer only explores one dimension, i.e., bandwidth.* ***Top****: we minimize cost such that happiness* ≥ 0.5*. We test 8 configurations (red) before convergence. The black (thick) line shows the optimizer's estimated happiness, and the shaded region its confidence. For under-explored regions, confidence is lower (i.e., a larger shaded region).* ***Bottom****: We can also incorporate a penalty for cost into the happiness function itself, and just maximize this function. This leads to a configuration similar to the top case also in 8 iterations.*

or reading objects in the cloud-managed DB service. We run the server on an Azure D4s v3 instance, with 4 vCPU cores and 16 GB of RAM. By running a client (on a separate instance of the same kind) that makes different types of queries to this Web server in different proportions, we can produce arbitrary workloads for the server.

We use a simple happiness function: the server monitors completion time for each request and reports the fraction of requests fulfilled in less than 2 seconds as the happiness index. For instance, if the server makes 300 requests since it last reported its happiness index, and 100 finish in under 2 seconds, the happiness index is 0.33.

Every time the server reports a new happiness value, our optimizer (emulating a cloud provider) runs its safe Bayesian optimization and identifies a new configuration to examine. The optimization goal we use is minimizing cost while keeping the happiness above 0.5. The tenant's default, overprovisioned configuration requests 80% of each resource. (This is a safe configuration, as it indeed results in happiness over 0.5 for each of our test workloads.)

A bandwidth-focused experiment: We configure the client to issue only bandwidth-focused queries. We allow the optimizer only to explore bandwidth provisioning, keeping DB and CPU fixed. Fig. 1 shows that the scheduler converges in 8 iterations without violating the SLA (*i.e.,* happiness ≥ 0.5).

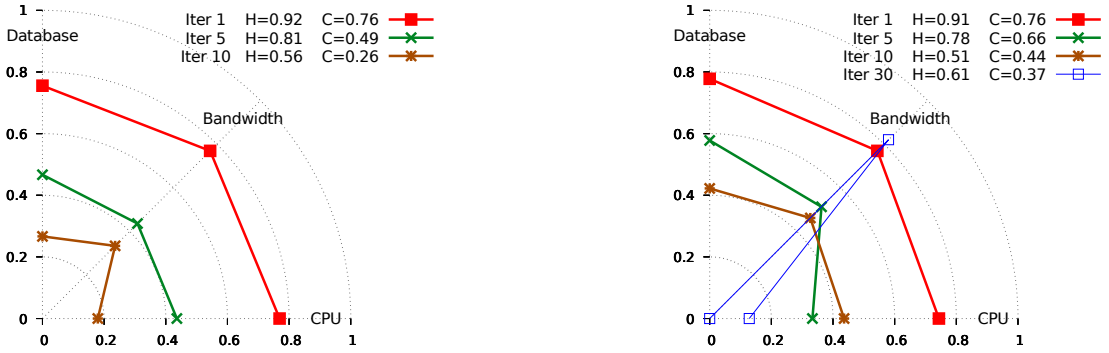

Figure 2: *The optimizer explores* 3 *different resources: network bandwidth, number of database request units, and maximum CPU utilization across two workloads: (a) a mixed workload, which uses all resources in a relatively balanced fashion; and (b) a bandwidth-focused workload. Each line represents one resource configuration. For the mixed workload, after* 10 *iterations, the optimizer converges correctly without violating the minimum happiness of* 0.5*. However, for the bandwidth-focused workload, the optimizer takes about* 30 *steps to converge.*

Figure 3: *The reported happiness can show substantial variation. Here we show the happiness reported across* 100 *measurements of the bandwidth-intensive workload and* 50% *available bandwidth. Note that each happiness value is computed over* 300 *client requests, and this does not eliminate the variation in this example.*

Each iteration consists of receiving one happiness index value and proposing a new configuration to evaluate.

happiness = f(performance, cost): Should we incorporate the cost directly into the happiness calculation? This is plausible, and the Bayesian optimizer still works. To illustrate that, we rerun the same bandwidth-oriented scenario as above, but this time with a modified happiness function that includes the cost (Fig.1). The new happiness function reports happiness as the inverse of the cost if at least 50% of all requests finish under 2 seconds. If not, happiness is 0. While we reach a similar bandwidth allocation, the search does not always satisfy the safety constraint that 50% of requests must perform above a certain threshold, because the function is not smooth.

Multi-dimensional optimization: We next test the same workload, but allow the optimizer to configure all 3 resources. As shown in Fig. 2(b), the optimizer again converges to the same provisioning for bandwidth as earlier (and lower provisioning for DB and CPU, which is desirable). However, the larger search space requires 30 iterations.

This simple setup also allows us to explore the impact of noise in the happiness values on the safe Bayesian optimizer. Fig. 3 shows that even for this simple setup, with a fixed rate of bandwidth-focused requests from the client, there is sizable variation in the reported happiness values. Nevertheless, the safe Bayesian optimizer converges without violating the target happiness constraint. We discuss the potential impact of greater workload instability in §6.

A more balanced workload: We also tested a more mixed workload comprised of an equal proportion of bandwidth, CPU and database requests. In this case, shown in Fig. 2(a), we observe that the optimizer discovers that the CPU and DB are more important than in the previous case. Convergence for this workload is faster, taking 10 iterations, likely due to the more similar impact of different resources.

4 Serverless and happy

Our exploration thus far has focused on traditional, long-lived tenant workloads, in a setting with virtual machines. However, the trend of serverless computing could potentially make an approach like ours even more attractive.

Serverless computing has unique properties suitable for our suggested approach. First, serverless functions are short-lived. Every sub-optimal choice thus lasts at most a few minutes[1]. For instance, consider a workload that incurs only 10% CPU utilization, but our optimization algorithm assigns one whole CPU core. If the workload lasts for one second, at current serverless prices, it would take more than 430 executions to incur an additional expense of 0.01$ due to this exploratory configuration. Second, the frequent feedback from short-lived functions implies a large number of optimization iterations with a low cost for exploration. Third, even the small burden of explicit feedback in terms of a happiness index may not be necessary for serverless functions. Cloud providers know the *execution time* of every function, which is, in many cases, a good proxy for performance and customer happiness.

[1]This depends on the cloud provider, but currently, 5 minutes is the maximum execution time for serverless functions across all major providers.

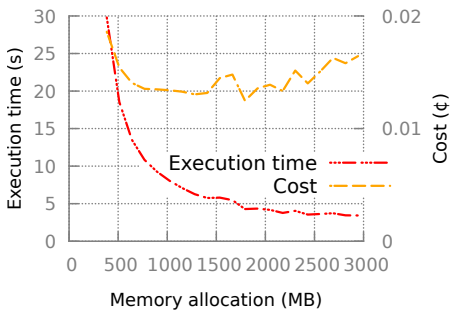

Figure 4: *Execution of one serverless function that performs image recognition using a TensorFlow model. As the resource allocation increases, the execution time improves, but the cost increases at the same time. To find an optimal configuration, the execution time is not enough – users must provide additional information about their budget.*

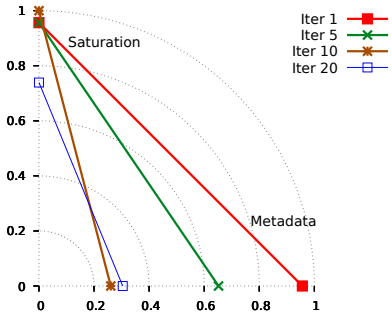

Figure 5: *Optimization of a video processing job that executes two serverless functions in parallel – one function extracts metadata from a video and the other applies color transformation on the video. Since the metadata function requires significantly less resources, the optimization algorithm has detected that successfully and decreased the resource allocation for the metadata function in order to decrease the cost.*

However, to run black-box optimization, we must know the cost constraints. Figure 4 shows the execution time and cost of a simple serverless function that loads one image and uses the image for object recognition using a model written in Tensorflow. As we increase available resources, the execution time decreases, but the cost increases. To find the optimal configuration, we must bound the cost, after which the optimization is simple.

Optimization of a single serverless function does not represent a serious challenge especially in the situation where we have information about the cost constraints. However, users do not always care about the performance of individual functions. Instead, a function represents a task in a larger job. In these cases, the execution time of one function might not be directly correlated to the execution time of the whole job. This problem can be mitigated in two ways. First, users can report their happiness as job execution time, or any other metric they want to optimize for. Second, the dependency between tasks can be automatically learned or collected from external services, *e.g.,* AWS Step Functions [2] that orchestrate the execution of multiple functions. From the obtained dependencies, we can directly infer the job execution time.

To illustrate the optimization process for a serverless job, we execute a simple video processing workload. For a given video file uploaded on the cloud storage (S3), we execute two lambda functions in parallel. The first function loads the file, changes the saturation of the video, which is a CPU intensive process, and stores the file back into storage. The other function more lightweight. It loads the video file, extracts basic video metadata, and stores it in a database.

Our goal is to minimize the cost of this simple video processing pipeline while keeping the job execution time below 20 seconds. The results are shown in Fig 5 [2]. We start by allocating one full core to each job[3]. In order to reduce the total cost, the Bayesian optimization allocates less resources over time to the lightweight metadata function, because its completion time does not influence the job completion, but can reduce the cost. The optimization converges in 20 iterations, after which the cost per execution is reduced by 15%.

Thus, our approach is potentially easily deployable and can yield immediate benefits in the serverless setting.

5 Conclusion

We explore a novel interface between cloud providers and their tenants that helps identify the best configurations for tenant applications. Our proposal, the happiness index, strives for simplicity, making it easy to use for tenants and providers; as well as efficiency, allowing the quick discovery of superior configurations.

While our results only scratch the surface of this problem, they indicate promise. For some scenarios, optimization based on the happiness index would be fast enough to be usable right away. Even a few hundred iterations could finish rapidly for applications like Web services, especially relative to the lifetime of applications in the cloud environment. Further, several inherent features of a serverless computing context could make our approach attractive there.

For more complex scenarios, particularly as the dimensionality of the search space grows, convergence to a suitable configuration could require more iterations. We next discuss this challenge among others.

Acknowledgments: We thank Sangeetha Abdu Jyothi, John Wilkes, Ganesh Ananthanarayanan, and the anonymous reviewers for their valuable feedback. We are also grateful to Google for their support in the form of a Faculty Research Award for Ankit Singla.

[2] Allocation 1.0 corresponds to 1536 MB on AWL Lambda. Allocation 0.0 means the minimum allocation of 128 MB

[3] 1536 MB of memory on AWS Lambda corresponds to one full CPU core.

6 Discussion Topics

Defining the happiness function: Cloud tenants could use any metrics their business depends on, *e.g.,* page load times for Web services, in framing happiness. Nevertheless, it could incur substantial effort to develop a happiness function over the many attributes the service's operators would otherwise manually look at. Clearly, poor design of the function could lead to arbitrarily bad results.

Scalability and convergence time: The search space for configurations increases with the complexity of tenant applications. If the tenant's happiness domain includes 100 components, then naively searching across even just 3 resources for each component entails a search space with 300 dimensions. However, for applications with large numbers of identical components (*e.g.,* a replicated Web service), the search space can be reduced to just the number of resources to explore. This intuition offers a simple default for the provider to explore, but can be used in more complex cases with a small amount of additional information from the tenant – explicit tagging of identical components (such as in AWS's "Auto Scaling Groups"). Nevertheless, the question of how many dimensions can be explored efficiently is, as of now, open.

Workload stability: As discussed in §3, the safe Bayesian optimizer can tolerate some noise, but we have not explored yet how much variation can be managed. There are two possibilities for addressing large variations. First, the tenant could ensure that they report happiness over periods long enough to encompass such variation. Second, such variations can be addressed via horizontal scaling (allocating more identical instances) or / and vertical scaling (increasing resource allocation for existing instances). It is possible that mechanisms like Amazon's Auto Scaling can be controlled by the optimizer in response to changes in happiness, but including this in our framework will require more work, particularly, addressing the sensitivity to variations – when is it appropriate to trigger horizontal or vertical scaling?

Tenant responsibility: Tenants must provide a safe start configuration, which meets their performance (or cost) target, as noted in §2. Only then can the provider search for configurations that optimize for minimizing cost (or maximizing performance). However, recent work indicates that most tenants today overprovision to meet performance goals [10], indicating that safe configurations would usually be available.

One caveat of our approach is that certain tenant applications may not benefit from dynamically increasing resource allocations. For example, an application that decides (at launch) how many threads to use based on the number of CPU cores available, will not benefit from the optimizer allocating more or fewer cores in its optimization process. We do not see an easy way to address this problem, and it may require additional logic in tenant applications of this kind, such that they can reevaluate hardware-based application logic periodically.

The provider's cost: We have so far used a very simplistic view of the cost of resources, assuming fine-grained linear behavior. While disaggregation in data centers could move us closer to this, at least for today's public cloud providers, we can expect that there could be a substantial cost to resource stranding from imbalanced use of resources. Obviously, the cost function the provider uses must account for such factors. However, in this regard, we note that some providers already offer dynamic pricing, and are moving towards a finer-grained implementation of the "pay for what you use" mantra, for example with "serverless" computing.

References

[1] AWS EC2 on-demand price list. https://aws.amazon.com/ec2/pricing/on-demand/. Accessed: 12-03-2019.

[2] Aws step functions. https://aws.amazon.com/step-functions/. Accessed: 12-03-2019.

[3] Azure virtual machine price list. https://azure.microsoft.com/en-us/pricing/details/virtual-machines/linux/. Accessed: 12-03-2019.

[4] Google Compute Engine pricing. https://cloud.google.com/compute/pricing. Accessed: 12-03-2019.

[5] Omid Alipourfard, Hongqiang Harry Liu, Jianshu Chen, Shivaram Venkataraman, Minlan Yu, and Ming Zhang. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *USENIX NSDI*, 2017.

[6] Felix Berkenkamp, Andreas Krause, and Angela P Schoellig. Bayesian optimization with safety constraints: safe and automatic parameter tuning in robotics. *arXiv preprint arXiv:1602.04450*, 2016.

[7] Felix Berkenkamp, Angela P Schoellig, and Andreas Krause. Safe controller optimization for quadrotors with gaussian processes. In *IEEE International Conference on Robotics and Automation*, 2016.

[8] Eric Boutin, Jaliya Ekanayake, Wei Lin, Bing Shi, Jingren Zhou, Zhengping Qian, Ming Wu, and Lidong Zhou. Apollo: Scalable and coordinated scheduling for cloud-scale computing. In *USENIX OSDI*, 2014.

[9] Eric Brochu, Vlad M Cora, and Nando De Freitas. A tutorial on bayesian optimization of expensive cost functions, with application to active user modeling and hierarchical reinforcement learning. *arXiv preprint arXiv:1012.2599*, 2010.

[10] Faruk Caglar and Aniruddha Gokhale. iOverbook: intelligent resource-overbooking to support soft real-time applications in the cloud. In *IEEE International Conference on Cloud Computing*, 2014.

[11] Archana Ganapathi, Yanpei Chen, Armando Fox, Randy Katz, and David Patterson. Statistics-driven workload modeling for the cloud. In *2010 IEEE 26th International Conference on Data Engineering Workshops (ICDEW 2010)*, pages 87–92. IEEE, 2010.

[12] Zhenhuan Gong, Xiaohui Gu, and John Wilkes. Press: Predictive elastic resource scaling for cloud systems. *CNSM*, 10:9–16, 2010.

[13] Herodotos H., Fei D., and Shivnath B. No one (cluster) size fits all: automatic cluster sizing for data-intensive analytics. In *ACM SoCC*, 2011.

[14] Jonas Mockus. *Bayesian approach to global optimization: theory and applications*. Springer Science & Business Media, 2012.

[15] Shivaram Venkataraman, Zongheng Yang, Michael J Franklin, Benjamin Recht, and Ion Stoica. Ernest: Efficient performance prediction for large-scale advanced analytics. In *USENIX NSDI*, 2016.

[16] Abhishek Verma, Luis Pedrosa, Madhukar Korupolu, David Oppenheimer, Eric Tune, and John Wilkes. Large-scale cluster management at Google with Borg. In *ACM Eurosys*, 2015.

[17] Neeraja J Y., Bharath H., Joseph E G., Burton S., and Randy H K. Selecting the best vm across multiple public clouds: a data-driven performance modeling approach. In *Proceedings of the 2017 Symposium on Cloud Computing*, pages 452–465. ACM, 2017.