AutoTune

Submission 116

Abstract

Most large web-scale applications are now built by composing collections (up to 100s or 1000s) of microservices. Operators need to decide how many resources are allocated to each microservice, and these allocations can have a large impact on application performance. Manually determining allocations that are both cost-efficient and meet performance requirements is challenging, even for experienced operators. In this paper we present AutoTune, an end-to-end tool that automatically minimizes resource utilization while maintaining good application performance.

1 Introduction

1.1 Motivation

Modern web-scale applications are increasingly built using multiple microservices (such as web servers, caches, load balancers, and data management systems), each of which runs in an isolated container [13] and interacts with other microservices over a virtual network. This approach allows operators to reuse microservices in several different applications, and to combine home-grown services with externally-provided ones. Companies such as Google [7], Uber [39], and Netflix [40] have deployed applications comprised of 100s or even 1000s of microservices.

To cope with this complexity, operators rely on *microservice orchestrators* such as Kubernetes [33] when deploying applications. Operators provide the orchestrator with an application specification – listing the required set of microservices, their resource requirements, and any placement constraints (*e.g.,* requiring that two microservices be placed on the same server) – and a cluster specification that describes the set of servers on which the application can be launched. The orchestrator uses these inputs to place and launch the microservices.

While these orchestrators have proven highly effective in automating the *management* of microservice applications, they do not optimize their *performance*. This is because the performance of microservice applications is a complicated function of the resources (such as memory and cores) allocated to each microservice (and, in some cases, the application performance also depends on microservice placement). Given the complexity of these large microservice applications, there is little hope of deriving analytical expressions for their performance (as a function of resources and placement). Instead, many operators merely overprovision with resources [7, 10, 11, 25] and use simple affinity rules for placement [3, 33, 37]. This results in an inefficient use of resources. We address this inefficiency with AutoTune, a tool for automatically reducing resource consumption while preserving adequate performance.

1.2 Considerations

Our choice of solution was strongly influenced by three basic considerations. First, we want a solution that can be used by non-expert operators, and is applicable across a wide variety of systems and use-cases. We therefore treat microservices as blackboxes, rather than assuming that detailed models of their performance are available. Similarly, we directly measure application performance rather than attempt to derive it from performance measurements on individual microservices. Moreover, we allow the operator to select the performance objective most relevant to them (makespan, tail latency, etc.), rather than designing AutoTune around a specific performance metric.

Second, we assume that operators have in mind (based on previous experience running the application) an initial deployment of their application that achieves acceptable performance. This deployment specifies the placement (on servers) and the resource allocations (*e.g.,* memory, cores) for each microservice. We take the performance of this initial deployment as the baseline performance expectation. Our goal is to find more efficient deployments which use fewer servers, yet achieve the roughly same level of performance (or better).

Third, in order to not interfere with the production environment, we intend AutoTune to be run on a testbed where various configurations can be tested. Cloud providers such as EC2 make such experiments feasible and orchestrators like Kubernetes make is relatively simple to deploy. In order for these testbed runs to be useful, we assume that the operator has access to a representative workload (or set of workloads) they can use for tuning application performance (see our later discussion in Section 3.5). This is most clearly true for recurring batch workloads, but it also applies to applications where the load is fairly predictable. We discuss extensions to more varying workloads later in the paper, but our initial focus is on predictable workloads. We also assume that placement is not a crucial determinant of performance, except in special cases where there is affinity (two microservices should be run on the same machine), so we focus on resource allocation as the most important factor in achieving good performance. However, as explained later, we do consider placement in situations where there is interference between two microservices.

1.3 Approach

Our approach to finding more efficient deployments involves two separate phases. In phase 1, which we call *resource clampdown*, we try to identify overprovisioned resources where we can reduce the resource dedicated to an individual microservice without impacting overall application performance. Once we have eliminated all overprovisioning, we then try to place microservices on the minimal number of servers while respecting these resource allocations. We then verify that the resulting deployment match the performance of the initial deployment, and make adjustments if necessary to make this hold.

In phase 2, which we call *performance improvement*, we take the deployment from phase 1 and test whether performance can be improved by assigning leftover resources and shifting resources on a server between microservices. The result is a locally optimal allocation of resources that roughly meets or exceeds the original performance goal.

We demonstrate the efficacy of our approach by using AutoTune on three representative microservice applications exhibiting a variety of microservice design patterns with widely different microservices (both third-party and custom). We demonstrate that, for these use-cases, AutoTune can reduce the number of servers by up to 6.6×, while simultaneously improving performance by up to 20%. Note that our goal in designing AutoTune was not to achieve *provably optimal* performance or *provably optimal* efficiency, but to provide a widely-applicable and easy-to-use tool that can provide *as good or better* performance than the operator-provided baseline while typically reducing the number of servers required.

1.4 Paths Not Taken

Here we explain why we did not use two particular approaches that might initially appear to be more natural choices. Operators often monitor the utilization levels (CPU, memory, disk, and network) in all the servers, and this can help them identify resources that might be performance bottlenecks. However, while such measurements are generally useful, high utilization levels do not necessarily indicate that a fully utilized resource is causing poor microservice performance. More generally, without an analytical model of application performance, one cannot hope that any set of local measurements can fully capture the impact of a particular resource or microservice on the overall application performance. We discuss these observations in more detail in Section 3.6.

In recent work [5], Bayesian optimization techniques were used to improve the performance of an application consisting of a single service (Spark executors) with a single microservice per service. Optimizing this problem merely requires determining resource allocations for that single microservice, leading to relatively low-dimensional search space on which Bayesian optimization performs well. In the setting we consider, an application consists of several distinct microservices, each of which has differing behavior and resource requirements, and these microservices interact with each other in a variety of ways, making it impractical to apply Bayesian optimization due to the high-dimensionality of the problem. We therefore turned to an approach that is more brute-force in nature, directly measuring application performance and varying resource allocations (and placement) to find deployments that minimize the number of servers used while preserving (or exceeding) the target performance. While this approach may lack intellectual elegance, it is both practical and easily-usable by non-expert operators.

2 The Design of AutoTune

2.1 Overview and Terminology

We begin the AutoTuning process by considering an application A, some performance metric P for that application, an initial deployment D_0 (defined by an initial resource allocation R and a set of placements), and a representative workload W. P_0 is the performance achieved on this initial deployment, and sets the operator's expectation level for subsequent deployments.

The microservice resources we consider are: memory, CPU cores (both full cores and fractional assignments), disk bandwidth, and network bandwidth. We use the term MR to refer to the allocation of one of these resources to a particular microservice on a particular server. AutoTune adjusts these resource allocations, and the placement of microservices, in order to minimize the number of servers required. This process consists of the two aforementioned phases, which we now describe in turn.

2.2 Phase 1: Resource Clampdown

In trying to reduce the number of servers required to adequately support the application A, it is crucial to identify any resources that have been overprovisioned (*i.e.,* more resources have been allocated than are needed to maintain the current level of performance) in the initial deployment. This is done by *stressing* the application by reducing an MR (that is, reducing the amount of some resource allocated to some microservice) and then measuring the resulting application performance. This allows AutoTune to identify which MRs impact application performance and which do not; we call these (respectively) *impacted microservice resources* (IMRs) and *non-impacted microservice resources* (NIMRs).

Phase 1 applies these stresses in an iterative fashion. In each iteration, AutoTune individually reduces the allocation of each MR until it identifies an NIMR (*i.e.,* until it identifies a resource that, when decreased, does not reduce application performance). It then reduces the resource allocation of this NIMR by the same amount it was stressed, and then repeats the iteration until no more NIMRs are detected. At this point, the resource allocation is *tight*, in that the reduction of any MR will reduce the application performance. At this point, we turn to seeking a better placement.

The process of stressing reveals how sensitive application performance is to each MR at the tight allocation. For brevity, we will say that an MR is more impacted than another if the application is more sensitive to that MR. For each microservice s we record which of its resources is most impacted, and will call this MR_s. In the placement phase we avoid placing two microservices with the same MR_s on the same server. When that is not possible, we then assign microservices in a round-robin fashion on the servers.

This process produces a deployment plan that packs microservices into a reduced number of machines based on measurements taken on a larger set of servers. However, containers do not provide perfect resource isolation, and when we increase the density of microservice placements the overall application performance can degrade. Thus, we deploy the packed placement to verify that the resulting application performance is close to P_0. If the packed performance degrades past a specified threshold, AutoTune generates variations on the placement. If these variations are exhausted without finding placement with sufficient performance, then a new placement is constructed in a deployment with additional servers.

2.3 Phase 2: Performance Improvement

At this point, AutoTune has a packed deployment (specifying resource allocations and placement) that achieves performance roughly equivalent to that of the initial deployment. In this second phase, we attempt to improve application performance by reassigning resources on each server using an algorithm that is essentially discrete gradient-descent.

Recall that the deployment provided by phase 1 may have unused resources on each server. We first assign these resources the microservice that is most sensitive to them (*i.e.,* to the most impacted MR of that type on that server); this information about sensitivity was discovered by the iterative process in phase 1. However, even when all resources are assigned, improvements are still possible. When two microservices α and β on the same server are both impacted by the same resource (*i.e.,* decreasing the amount of that resource for either microservice reduces the application's performance), the application's performance might still be improved if adding some amount of that resource to α helps the application's performance more than removing that amount from another β.

Listing 1: Algorithm for Gradient Descent from Performance Improvement Phase. The algorithm starts from the placement and resource allocation determined in Phase 1, and then adjusts these allocations based on transfers. The function Gradient(R) produces a vector of changes in performance based on discrete changes in each MR (where the step size is δ_r for resource r), and the sort function orders them in decreasing order.

```
Given:
  Starting Resource Allocation: $R$
  Performance Function: $P$
  Step size: $\delta_r$ for resource $r$

do:
  PrevP = $P(R)$
  max_impact_list = sort(Gradient(R))
  min_impact_list = reverse(max_impact_list)
  transfer_found = False
  for m_h in max_impact_list:
      for m_l in min_impact_list with m_l < m_h:
          if MRs are colocated,
              and have same resource type $r$:
               R[max_IMR] += $\delta_r$
               R[min_IMR] -= $\delta_r$
               transfer_found = True
               break
      if transfer_found:
          break
  CurrentP = $P(R)$
while (CurrentP - PrevP > threshold)
```

We use a gradient descent method to look for these transfer opportunities, via the algorithm described in Listing 1. This algorithm starts by listing all MRs in the order of their sensitivity then being an iterative process, starting with the most sensitive MR, of trying to match highly sensitive MRs with less sensitive MRs colocated on the same machine and having the same resource type.

3 Additional Issues

We will describe the implementation of AutoTune in Section 4, but we first address a set of additional issues that are relevant to our design.

3.1 Minimizing Redeployments

One factor that heavily influenced our design is that every time one changes the placement, one needs have a warm-up period so that various caches can reflect steady-state performance. To avoid this extra overhead, we chose an approach that, in the typical case, only looked at two placements: the initial deployment, and the placement that is produced at the end of phase 1. Note that the stresses that occur in phase 1 only decrease resource allocations, and the transfers that occur in phase 2 do so in a way that are feasible within a single server. Thus, the only change in placement needed is at the end of phase 1 (assuming that the deployment that arises out of the clampdown process has sufficient performance), so there are only two warm-up periods.

3.2 Discrete Gradient Descent

Analytically, gradient descent uses derivatives in order to choose the direction for change. Here, we must implement a discrete derivative (when stressing the application); in order to evaluate the derivative, we compare the application performance with the original MR with one that differs by an amount δ, and the question is how do we choose δ. AutoTune sets δ proportional to the server's resource capacity. However, rather than use a fixed percentage, we start at 30% and then, at each iteration of the algorithm, we decrease this percentage by a factor of 1.2. This allows us to take finer and finer tests of discrete changes. We also implement a binary search feature when evaluating resource transfers. When we one MR by δ and decrease another by the same amount and find no change in application performance, we then execute a binary search between the original and tested allocations to see if we overstepped a local optima.

3.3 Reducing Search Space

If we were to investigate the impact of changing every resource individually, we would have to investigate $4M$ different possible changes in MRs, where M is the number of microservices (and 4 is the number of resources per microservice). To cut down on this search space, we begin by randomly dividing the search space into p partitions. Within each partition, we simultaneously reduce the allocation of *all the MRs*, and measure the performance impact of this cumulative reduction in resources. Each AutoTune phase uses partitioning differently:
Partitions in Resource Clampdown By default, AutoTune randomly separates the MRs into two partitions. If the performance does not degrade, then it can directly inferred that *all MRs in that partition can be safely reduced*, and conversely if it does degrade, we cannot reduce any of the MRs at that stage in the clampdown process. Once we reach the point where no partitions can be reduced, we then increase the number of partitions allowing finer grain experimentation.
Partitions in Performance Improvement When using gradient descent, AutoTune identifies the partition that is most impacted. Within that partition, AutoTune applies the original gradient descent method, stressing each MR individually. Thus, in each iteration of the the gradient descent process, AutoTune with the pruning method with p partitions only needs to explore $\frac{N \cdot k}{p} + p$ MRs per iteration, rather than of having to explore $N \cdot k$ MRs.

Overall, our experience with partitioning is that it reduces our runtime by a factor that varies between 2 and 4.

3.4 Placement

In the introduction we made the assumption that placement was not a crucial factor in application performance aside from special case of affinity, yet we later mention that the lack of complete isolation means that placement can lead to interference. Interference in AutoTune can result in performance worsening during the placement phase of resource clampdown (§2.2). We resolve this by adding an additional placement constraint, where we try to preserve the colocations that were present in the initial deployment; that is, if microservices α and β were on the same server in the initial deployment, we try to maintain that colocation as we reduce the number of servers (unless doing so would result in us violating the placement condition in §2.3). We do not try to explore the space of potential placements more broadly, as that would greatly increase the search space. Furthermore, independent of whether interference is observed or not we obey all user provided placement constraints (such as about affinity, or which server-types microservices can run on) when placing services.

3.5 Dynamic Workloads

We started by focusing on applications with relatively predictable workloads, and assuming that the operator supplied a representative workload which could be used to evaluate performance. Since almost all workloads have some natural variation, we further assume that this supplied workload represents the high-end of anticipated workloads (and the degree of such overestimation is up to the operators based on their tradeoff between performance and efficiency). Moreover, the initial deployment supplied by the operator should have acceptable performance on this workload.

However, we can also generalize AutoTune to more dynamic workloads, where the variation is too great to account for with the above approach. We require that the production version of the application have real-time monitoring *application performance* and *per-microservice traffic volume*. We consider four cases:

Constant Performance, Increased Traffic Volume: This suggests that the application remains in a locally optimal regime even as traffic load increases. In section §6, we demonstrate that resource allocations determined by AutoTune continue to provide comparable performance despite small workload fluctuations.
Constant Performance, Decreased Traffic Volume: In this case, the AutoTune resource allocations might be overprovisioned. If the operator gauges that the decrease in traffic volume is both meaningful and sustained, she may opt to rerun AutoTune with this decreased workload.
Worse Performance, Increased Traffic Volume: In this case, we do not change the allocations of the existing containers (since this would require redeploying some them); instead AutoTune activates horizontal scaling (commonly deployed with Kubernetes [2]) to deploy more instances. At stability, the operator may elect to retain the workload traffic and re-run AutoTune on this new workload.
Worse Performance, Constant or Decreased Traffic Volume: Here there may be external factors influencing per-

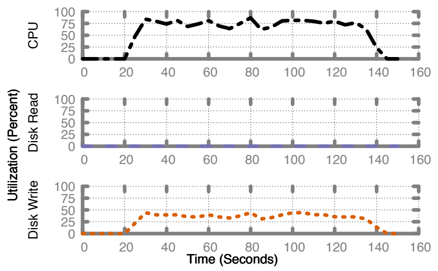


Figure 1: Observed utilization for frontend microservice.

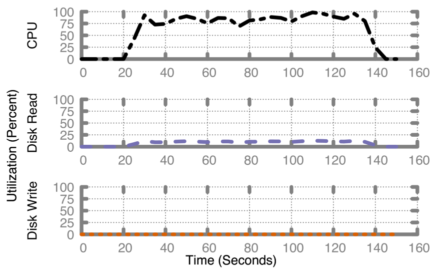


Figure 2: Observed utilization for storage microservice

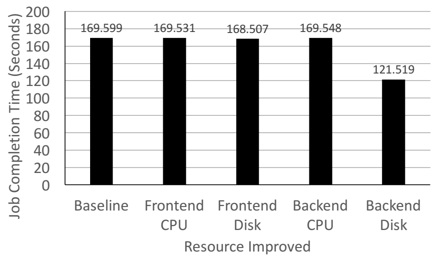


Figure 3: Actual Job Completion Time after adding resources to the multiservice microbenchmark.

formance, or the workload might have changed in ways not visible to the monitoring (*e.g.,* the nature of the queries may have changed to be more compute intensive). If the problem persists, the operator may elect to retain the workload traffic and re-run AutoTune on this new workload.

3.6 Why not just measure utilization?

A common first reaction to hearing about AutoTune is: *why don't you just measure utilizations?* Here we illustrate why that simple, and often effective, approach is not sufficient. For the following scenario, we ignore memory allocations and only consider the allocation of three resources: CPU cores, network bandwidth and disk bandwidth.

Consider an application comprised of two microservices: a frontend microservice and a storage microservice. The frontend microservice receives client requests and performs a blocking remote read from the storage microservice, which on receiving a read request, reads and processes the request file before returning the processed results. We collected resource utilization and job-completion times for each microservice during a baseline run, which we show in Figure 1 and Figure 2. Utilization seems to indicate that bottleneck resource is either (a) the CPU on the frontend microservice; (b) the CPU on the storage microservice; or (c) the *disk* on the frontend microservice (used for logging). We then attempt to empirically validate these findings by increasing each of these resources and measuring their impact on the application's performance (Figure 3). We find, contrary to what the utilization measurements suggest, that none of them has any impact on application performance. In fact, the highest performance improvement comes from increasing *disk bandwidth* on the storage microservice, a resource that was not highly utilized! This is because the frontend is blocked while waiting for the storage service to read data, and utilization is not sufficient to capture this dependency. Thus, in answer to the oft-asked question about utilization, we answer that there must be a tool that contextualizes the importance of a particular microservice and resource to overall application performance, and AutoTune does to by measuring this performance directly.

4 Implementation

4.1 Integration with Container Orchestrator

AutoTune can be integrated with any container orchestrator that exposes microservice placements, and then run at the push of a button. Our current implementation has been integrated with two major orchestrators: Kubernetes and Kelda [23]. AutoTune is packaged inside a Docker image and deployed as another service, and operators can use it by importing AutoTune.

4.2 Other Inputs

Operators using AutoTune must also specify a configuration generator for each microservice and a performance metric. In our current implementation, operators implement the configuration generator as a Python class which implements a method that accepts as input the current resource allocation, and outputs an appropriate configuration. We expect that in many cases this configuration script would just call existing utilities such as `pg_tune` [31] to generate and appropriate configuration. Operators also implement the performance measurement function (P above) as a new service, which AutoTune can invoke when measuring application performance. For our evaluation we relied on application-specific load generators, *e.g.,* ApacheBench [4].

4.3 Stressing

AutoTune stresses (*i.e.,* throttles) a microservice's resource allocation in its pursuit of a more efficient deployment. While our current design only stresses four resources – CPU core allocation, CPU quota per core, network bandwidth, disk bandwidth, and memory allocation – there are several other resources that could be stressed in modern processing units (e.g., memory bandwidth, disk latency, etc.); AutoTune can be easily extended to allow stressing of other resource and AutoTune is designed to allow additional stressing mechanisms to be plugged in. Additionally, the exact method of stressing also depends on the configurations of the underlying operating system. It is important to ensure that all the stress mechanisms are work conserving. Currently, our implementation and evaluations are all conducted using a standard Ubuntu 17.04 image from the AWS AMI Marketplace. Differently configured operating systems running in different domains may require the user to modify stressing mechanisms.

Below, we briefly describe how we stress the resources:

CPU Quotas: Linux Control Groups, or `cgroups` [32] are a kernel feature that enables resource allocation. AutoTune currently makes use of two mechanisms to throttle CPU allocation for a particular service. First, AutoTune can dictate the proportion of CPU time that CFS allocates to a container by setting a container's period and quota. The current implementation allocates a quota relative to a fixed period (chosen at the start of the experiment) such that the maximal stressing never exceeds the minimal scheduling time quantum in the underlying operating system.

CPU Cores: While there are ways to fix processes to cores (e.g., using `cpuset`, the effectiveness of allocating more cores to a particular process is contingent on that process being able to fully utilize the provisioned cores. This is typically dictated by a knob located in the software configuration of a particular service. For example, Spark exposes a variable that specifies the number of driver or executor cores. Due to the nature of this particular kind of stress, changes in CPU core provisions is achieved by jointly tuning the software configuration (defined a priori to allow the application to leverage more cores) and the hardware provision. The corresponding hardware improvement is achieved by provisioning a larger CPU Quota to the service such that the amount of CPU Quota per software core is fixed. More precisely, given a service with *quota* aggregate CPU quota and c cores, throttling down by one core would mean provisioning a quota of $\frac{quota}{c} \cdot (c-1)$. Subsequently, we rely on the CPU scheduler to schedule this smaller quota over fewer physical cores.

Memory stressing: AutoTune stresses memory through the `cgroups` memory subsystem. Each service is allocated some memory capacity, which would include memory capacities provisioned over the hardware memory and swap space. In order to prevent bricking the application, AutoTune will monitor the memory utilization of the service to ensure any stress applied to it will not cause the application performance to fail (e.g., OOM).

Disk Bandwidth: AutoTune stresses the disk using `blkio` [32] (again through `cgroups`), which allows for hard limits on both the read and write from specific block devices. AutoTune allows for both joint and individual throttling of read and write. In our experience the `blkio` subsystem imposes limited CPU overheads.

Network Bandwidth: AutoTune stresses the network by limiting link bandwidth using a standard Linux based tool, `tc` [22]. To do this we first measure the maximum attainable inter-VM network bandwidth, and then impose $k\%$ stress by limiting the container's bandwidth to $(100-k)\%$ of this maximum. `tc` uses hierarchical token bucket (HTB) to implement this rate limit, and scheduling network traffic using HTB imposes some CPU overhead. In our experience this additional overhead did not noticeably affect our results.

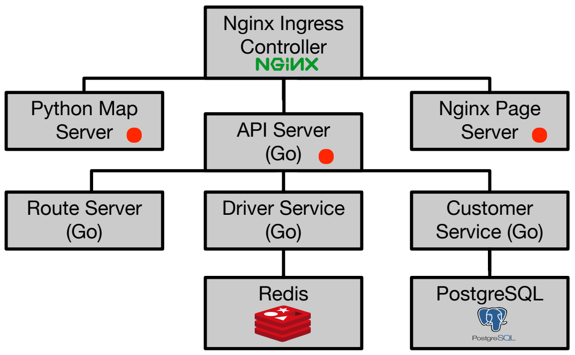


Figure 4: HotROD Application, modified from Uber. Red circles indicate endpoints. Not shown: HAProxy load balancers between the ingress and the map and API endpoints.

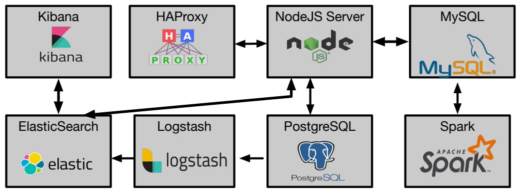


Figure 5: Apartment Rental Multi-service Application

5 Microservice Application Overview

We sought to create microservice applications that mirror characteristics of commercial infrastructures (here we focus on deployments commonly seen in enterprises and other moderate scale deployments, rather than on services such as S3 and Google Search which are designed to support simultaneous access by a large fraction of humanity and can afford to hire experts to analyze and finely-tune their deployments). We begin by introducing some common microservice design patterns and microservices that are commonly deployed together. Finally, we present three end-to-end microservice applications that we deployed AutoTune on.

We surveyed a number of industry practitioners to determine (i) which open source microservices were commonly used together, e.g., ELK (Elasticsearch, Logstash, Kibana) stack, Spark/MySQL, etc. and (ii) how these services are architected, e.g., chain or aggregation patterns [18] [34]. Ultimately, we designed the following applications.

HotROD Application (30 MR): diagrammed in Figure 4 is a (modified) distributed tracing application originally created by Uber to demonstrate its open source tool Jaegertracing [35]. We modified HotROD slightly by first splitting up the frontend to be a separate API service and Nginx service to serve static pages. We also added a mapping service to pull graph data from S3 which is plotted using Networkx [21] and returns a JSON file which is rendered on the webpage. An Nginx ingress and some Haproxy load balancers where also used to distribute load across all replicas of the mapping and API service.

Apartment Rental Application (48 MR): We use an application we developed for testing our project that is comprised of eight services. The application exposes six different endpoints, and we rely on a load generator that simultaneously calls to all these endpoints. The performance metric is a linear combination of the the 99th percentile latency across these endpoints. We show the services that comprise this application, and the dependencies between these services, in Figure 5. Every request first arrives at the Node web server, and then traverses up to nine other containers during the request's lifetime (i.e., requests through the Elasticsearch cache that retrieves table entries from PostgreSQL through Logstash upon a cache miss). Certain microservices simultaneously serve requests from two endpoints, as in the case of PostgreSQL. We test this application with two types of workloads: a write heavy workload (referred to as *Apartment App Write*) and a workload consisting of a mix of reads and writes (referred to as *Apartment App Mix*).

MEAN Stack (12 MR): The MEAN stack is a popular Javascript software stack comprised of three heterogenous services with distinct functions: MongoDB (database service), NodeJs (web server), and HAProxy (load balancer). Our MEAN stack deployment consists of a *Todo Application*, which effectively allows an user to Put, Get, and Delete elements. The workload consists of writes and reads to the frontend; each request traverses all microservices in the application.

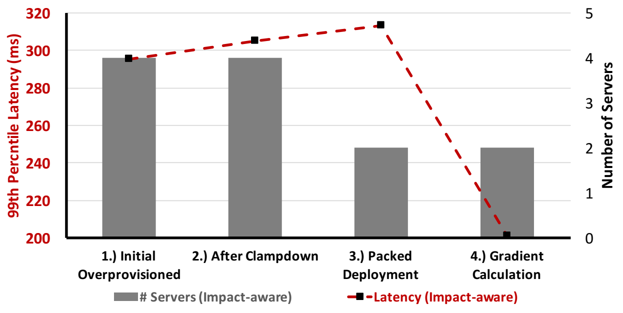
6 Evaluation

We begin by evaluating the overall effectiveness of AutoTune by demonstrating that running it on deployments of the applications described in §5 results in a reduction in the number of servers required and leads to improved performance. Next we demonstrate the robustness of our techniques by evaluating the impact of factors such as the original deployment provided as input, the presence of noisy-neighbors, and workload variations on our result. Finally, we use microbenchmarks to evaluate the effectiveness of the individual techniques described in §2 and §3.

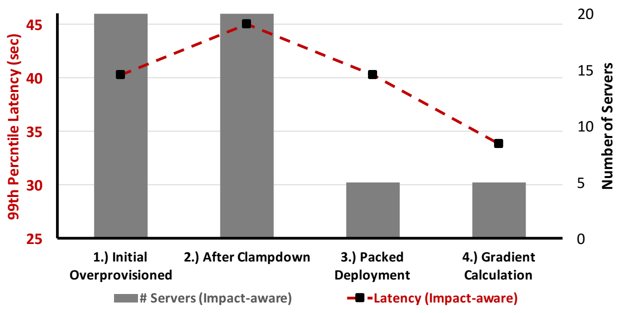
Unless otherwise specified all of the evaluation in this section uses 99th percentile tail latency as the performance metric to be optimized. We have tested AutoTune with other metrics, and obtained similar results but omit these for brevity. Finally, unless specified, we ran our evaluation using general purpose *m4.xlarge* instances on AWS EC2.

6.1 End-to-end Experiments

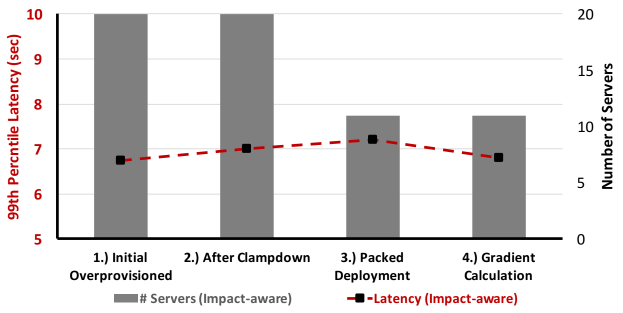
We begin our evaluation AutoTune by performing an end-to-end test and measuring its effectiveness. We show results for the MEAN stack in Figure 6a, the apartment application with mixed workload in Figure 6b, and the HotROD application in Figure 6c. In these graphs, we show how performance (99th percentile latency) and server usage changes as AutoTune runs. We show performance (line) and number of servers (bar) at four points: (1) the initial overprovisioned deployment, (2) after stressing and reducing each microservices resources (but before changing placement), (3) at the end of phase 1 (§2.2)



(a) MEAN Stack



(b) Apartment Application



(c) HotRod Application

Figure 6: Changes in Resource Efficiency (bars) and Performance Improvements (lines) over the course of running AutoTune end-to-end.

after microservices have been packed onto fewer servers, and (4) at the end of the performance improvement phase (§2.3).

In the three applications we evaluated, the initial deployment split resources evenly between the three microservice instances. After running AutoTune, the MEAN stack achieved 33% better performance with half the number of servers, the apartment application achieved 20% better performance with 2.5× fewer servers, and the HotROD application used half the servers with roughly the same performance level. In order to determine these final deployments, AutoTune required roughly 13, 18, and 18 hours for MEAN stack, Apartment App, and HotRod App, respectively.[1] While the exact improvements vary based on application, we see significant improvements in all for resource efficiency, and performance that

[1]These times might seem long, but we run every trial up to 25 times in order to reduce fluctuations in the results.

is *at least* as good as the initial deployment – in a reasonable amount of time.

The graphs also show that the MEAN stack and the write heavy workload of the apartment application experience slight performance degradation in the earlier stages of the process, which was then corrected by the performance improvement phase. This is because the available network bandwidth between microservices increases when they are packed onto fewer machines, and this shifts the bottleneck (in this case to the CPU). The Performance Improvement Phase can hence correct for this problem through resource transfer. While AutoTune was able to improve the HotROD application's resource efficiency, observe that performance does not improve in the Performance Improvement Phase. The HotROD application contains 3 separate microservices that are all compute bottlenecked; consequently, in the clampdown deployment, there was very little free resources for impacted MRs to grow into. Nevertheless, AutoTune was able to maintain the performance of the application. This is in contrast to the Apartment Application which exhibits a wide variety of bottlenecked resources (CPU, Disk, and network); this diversity enabled performance improvements in the Performance Improvement Phase.

Since we cannot compute the optimal placement and resource allocation without an exhaustive search, we use the initial (input) deployment as our baseline. As we previous discussed (§2), AutoTune merely seeks to ensure performance remains as good as this initial input.

6.2 End-to-End Robustness Considerations

Next we look at AutoTune's robustness to variations in user input and deployment conditions.

6.2.1 Robustness to Initial Deployment

Recall that AutoTune identifies locally optimal resource allocations; thus the operator-provided initial deployment can have a significant impact on the outcome. In the end-to-end experiments shown above (6.1), the initial deployments placed three microservice instances on each server, where each microservice instance has an equal split of the server's total resources. We look at three different sets of initial deployments. 1.) deployments where each microservice instance has been provisioned the maximal amount of resources for the instance type 2.) *randomly generated* deployments and 3.) deployments that result from horizontal autoscaling.

Maximally Provisioned Initial Deployment: In this initial deployment, each microservice instance was provisioned an entire EC2 VM instance. In doing so, AutoTune continued to demonstrate effectiveness in reducing server usage. AutoTune was able to reduce server usage by 75% for the Apartment App and the HotRod App by 40%. However, AutoTune did not register performance improvements because IMRs were already maximally provisioned; during the performance improvement phase, there were no resources to transfer because the impacted MRs were already maximally provisioned. Compared to the initial deployments described in the end-to-end experiments (with 3 per machine), AutoTune identifies a deployment that is more performant but also more expensive. For instance, for the HotRod application, starting from a maximally provisioned initial deployment results in a final deployment that requires 2.5x more servers, but also achieves 2.2x better performance.

Randomized Initial Deployment: Table 1 demonstrates four such random deployments with the Apartment App. Note that in all cases, AutoTune was able to cut resource usage by 57% – all while either improving or maintaining performance. Interestingly, regardless of the initial performance, 3 out of 4 of the random initial deployments converged to the same local optima. In the one auspicious scenario where the initial random deployment resulted in improved performance, AutoTune was able to converge back to that same performance. For brevity, we elide results from other applications.

Random Dep. #	# Server Reduction	Init. Perf. (s)	Fin. Perf. (s)	% Improve
1	57%	68.3	45.2	33.8
2	57%	38.2	38.7	-0.01
3	57%	63.4	47.0	25.9
4	57%	58.5	47.9	18.1

Table 1: End-to-end runs of Apartment Application App with randomly generated initial deployments

Horizontal Autoscaling: In Section 3, we discussed how AutoTune could be used jointly with dynamic workloads. In this section, we presented four possible scenarios. We do not explore the scenarios where a workload change does not affect performance, as this typically does not require operator intervention. In this section, we step through the AutoTune-based solution where increased workload results in performance degradation.

Upon a workload change, resource allocations revert to being overprovisioned (*i.e.,* the initial deployment) and will horizontally scale until some threshold in the scaling policy is met. When the elevated workload stabilizes, the operator can optionally deploy AutoTune using that new workload. From a performance and resource efficiency point of view, what does the operator potentially have to gain from switching to AutoTune-calculated resource allocations at that workload? To this end, we deployed all three of our microservice applications on Kubernetes, and enabled horizontal autoscaling (HPA). HPA in Kubernetes implements a control loop that checks the average percentage utilization across all pods, and scales up if that threshold is exceeded. We ran our experiments across two separate utilization thresholds: 10% and 90%.

To simulate a sudden burst in load, we increased the workload by a factor of 10x. Upon observing the performance degradation as a result of the increased workload, the horizontal autoscaler would first take over, switching to an overprovisioned resource allocation. Rather than falling back to this overprovisioned deployment everytime, the operator could record the increased workload and subsequently run Auto-

Application	90% Scaling Threshold			
	w/o AutoTune		w/ AutoTune	
	Servers	Performance	Servers	Performance
MEAN	11	1668	6	1630
Apartment App	21	7919	13	7615
HotRod	22	20781	18	21302

Table 2: Server usage and Performance resulting from horizontal autoscaling, compared against use of AutoTune. Horizontal Autoscaler set to scale when utilization crosses 90%

Application	10% Scaling Threshold			
	w/o AutoTune		w/ AutoTune	
	Servers	Performance	Servers	Performance
MEAN	29	1141	14	1188
Apartment App	25	7686	22	6158
HotRod	–	–	–	–

Table 3: Server Usage and Performance resulting from horizontal autoscaling, compared against use of AutoTune. Horizontal Autoscaler set to scale when utilization crosses 10%

Tune on that higher workload.

The comparison between the horizontal autoscaling approach and AutoTune are in Table 2 for the 90% scaling threshold and in Table 3 for the 10% scaling threshold. Performance ranged from very slight decreases to 20% increases, while the number of servers decreased between 12% and 52%. We omit the result of HotRod with 10% scaling because it scaled aggressively beyond the capacity of our resources.

6.2.2 Robustness to Noise

One commonly cited issue in multitenant cloud environments is the noisy neighbor issue, i.e., when different tenants scheduled on the same physical server create performance interference. While there are ways to ensure that interference does not occur (i.e., reserving dedicated servers), this is often prohibitively expensive. Note that our complete evaluation was conducted on standard EC2 instances (m4 instances) with no special isolation guarantees. We observed very little variability in our experimental results between different runs.

For a more structured analysis, we deployed our application containers alongside containers running computation at random intervals. This emulates the effect of a multitenant environment. The noisy neighbors run workloads that use all resources *i.e.,* CPU, disk, memory and network. To emulate multitenant environments, these neighboring containers are pinned to a different core from the application under test.

Table 4 shows the server reductions across deployments with and without noise. Server reductions remain unchanged while performance improvements varied somewhat (in both directions). Note that the results in this table are deployed on a different instance type (with more resources to facilitate the noisy neighbors) and should not be compared with earlier end-to-end experiments.

6.2.3 Robustness to Workload Fluctuations

AutoTune's mission is simple: given an initial deployment and a representative workload, AutoTune tries to find a more efficient deployment (fewer servers) with at least as good performance. One might worry that after this process of reducing

Application	% Server Reduction	% Performance Improvement
Apartment	57	14
Apartment (w/ noise)	57	10
HotRod	42	4.6
HotRod (w/ noise)	42	1
MEAN	50	4
MEAN (w/ noise)	50	17

Table 4: Comparing AutoTune's efficacy towards server reduction and performance increase with and without noise.

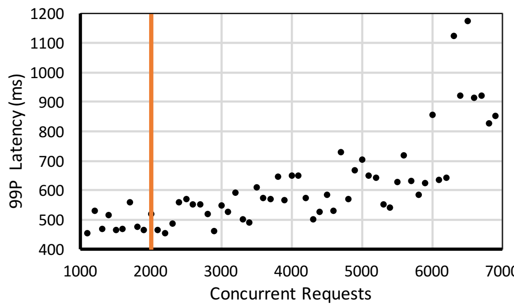


Figure 7: Robustness of MEAN App performance with additional concurrent requests. AutoTune used a workload of 2000 requests, indicated by the dashed vertical line.

servers, the resulting deployment would be so highly tuned that it would become very sensitive to workload fluctuations. This would be bad, since fluctuations are inherent in customer-driven applications. Fortunately, in our experience of running the applications we discuss here, the resulting performance is not highly sensitive to small workload fluctuations. We illustrate this with an example.

We took the MEAN application, and then altered the workload by increasing the rate of requests. Because the clients operate in closed request loops (request, then response, then new request), we merely increased the concurrency of this loop to issue several simultaneous requests. The original workload had a concurrency of 2000, and Figure 7 shows how MEAN application performance changes as this level of concurrency is varied, while using the deployment computed by AutoTune on the original workload. As shown in the graph, 99p latency grows gradually until the number of concurrent requests triples (from 2000 to 6000 concurrent requests), at which point the performance starts degrading far more rapidly. At this point, the operator would benefit from rerunning AutoTune with a workload consisting of 6000 concurrent requests or more.

This fits with our general experience with AutoTune that application performance does not suddenly degrade. One reason for this is that while the clampdown process produces a tight resource allocation, once these allocations are used to produce a new deployment, there are excess resources on each server because one typically cannot perfectly binpack the microservices into a minimal set of servers. In addition, applications are typically designed to degrade gracefully rather than fall off a cliff, and this might provide another reason for resilience.

In addition, recall that we assumed that the representative workload was supposed to be on the high-side, to directly pro-

Application	Initial Servers	Final Servers	Iterations	Clampdown Hours
Apartment App Write	21	3	3	1
Apartment App Write	7	3	4	1
Apartment App Mix	21	5	8	3.5
Apartment App Mix	7	4	4	1.75
HotRod	21	11	9	5
HotRod	7	4	6	4.4
MEAN Stack	4	2	4	1

Table 5: Running just Resource Clampdown on applications across two different initial deployment settings.

Application	% Improve	Time (hr)
Apartment App Mix,99p latency	26	11
Apartment App Mix, 50p latency	30.4	16
MEAN Stack, 99p latency	65.9	9
MEAN Stack, 50p latency	45	10
HotRod, 99p latency	6	12

Table 6: Overall Performance gains from gradient step in performance hill climbing.

vide some buffer from workload variations. Thus, between the inherently conservative nature of the representative workload, and the observed resilience against moderate load increases, we are less worried about AutoTune being overly susceptible to workload fluctuations.

6.3 Resource Clampdown Phase Illustration

Now that we have examined AutoTune's overall impact in terms of server reduction and application performance, we focus in on the two phases of AutoTune to see where these improvements come from. In this subsection, we demonstrate the resource efficiency gains and runtime of just Resource Clampdown Phase. Table 5 details the server savings as well as the Clampdown runtime. In keeping with the previous sections, we provide two initial deployment configurations: one container per machine, and three containers per machine. Only one initial deployment is shown for MEAN stack, since the MEAN stack only has four microservice instances. We find that in all cases resource utilization reduces at the end of this phase, and that across applications this phase took between 30 minutes and 7 hours. This time depends on several factors including the input deployment and the chosen workload, and can hence be reduced by having the operator make appropriate choices.

6.4 Performance Improvement Phase Illustration

In this subsection, we evaluate the second phase of AutoTune: Performance Improvement. Like the previous section, we start by providing an overview of its performance gains and runtime. Rather than just stopping there, we go a step deeper into the two primary mechanisms that underlie the Performance Improvement Phase. First, we evaluate the effectiveness of resource stressing for IMR identification. Unlike the Resource Clampdown Phase, where we directly measure the performance impact of reducing an MR, the Performance Improvement Phase requires *inferring* the outcome of increasing a resource, rather than directly evaluating the performance at the higher level. We then provide an example of how resource transfer works.

6.4.1 Performance Improvement Phase: results

We start by providing overall results from applying Performance Hill Climbing to these applications, before evaluating the individual steps in this phase. Table 6 shows the performance improvements in this phase, showing that Performance Hill Climbing in these cases improves application performance by between 17% to 66%. While achieving this improvement takes several hours for some applications, this is determined by the performance metric and workload selected by the operator, and hence depends on the setting in which AutoTune is used.

Application	Single MR	% Improve	Most impacted?
Apartment App Write	Node, CPU	30.4	Yes
Apartment App Mixed	Node, CPU	16.1	Yes
HotROD App.	Api, CPU	16.1	Yes
MEAN Stack	Node, CPU	55.5	Yes

Table 7: Effect of Provisioning more resources to the Most Impacted MR on the first iteration of the Performance Hill Climbing Phase. Larger performance gains are achieved over multiple iterations (*Table 6*).

6.4.2 Performance Improvement Phase: Stressing Effectiveness

We dive a level deeper and explore the inner workings of AutoTune, starting with the fundamental mechanism of AutoTune: resource stressing. The ability to determine the relative sensitivity of an MR underlies AutoTune's effectiveness. This is true for both phases of AutoTune. We do not evaluate the stressing mechanism for the Resource Clampdown phase because we do not need to *infer* anything about the MR. Stressing a resource is precisely equivalent to clamping down a resource, so there is no possibility of false positive in identifying IMRs or NIMRs.

On the other hand, false positives are possible in the Performance Improvement Phase. While we empirically have found that AutoTune does not require an exact, complete ordering of MR sensitivity, the algorithm revolves around two basic tasks that are important to get right. The first is identifying the most impacted MR, as it is the most direct way of improving AutoTune performance in the Performance Improvement Phase. The second is to distinguish impacted MRs from non-impacted MRs, since we only want to transfer resources to impacted MRs. False positives in IMR identification are possible because we are indirectly inferring a microservice's sensitivity to adding a resource from its sensitivity to reducing that same resource . In this subsection, we illustrate how effectively AutoTune accomplishes these two goals for the applications we tested.

First, we demonstrate that resource stressing is an effective way of identifying the most impacted MR. Table 7 shows that in every application that we tested, AutoTune was able to identify the most impacted MR correctly. We confirmed that the MR was indeed the most impacted by exhaustively increasing each MR's resource allocation and measuring the

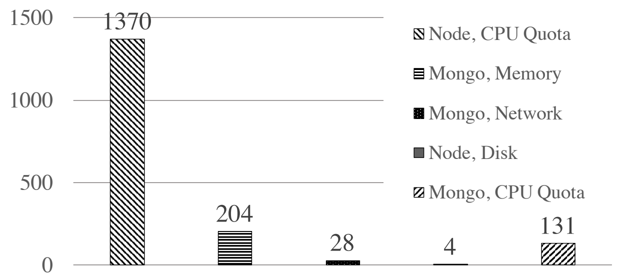


Figure 8: MEAN Stack Performance Degradation upon Single Resource Gradient (99^{th} percentile latency)

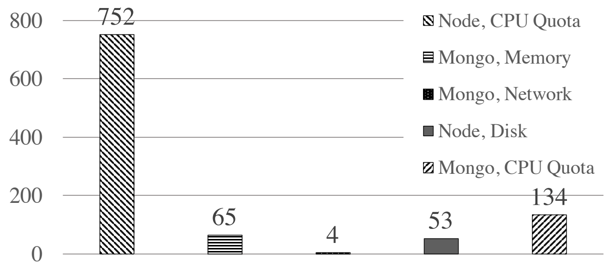


Figure 9: MEAN Stack Performance Improvement upon actual improvement resources (99^{th} percentile latency)

resulting performance.[2]

Next, we look at AutoTune's ability to positively identify an impacted MR. We illustrate IMR identification with an example using the MEAN stack. Figure 8 shows the measured end-to-end performance resulting from stressing the a single MR. Note that we omit some MRs from the figure for clarity; the MRs we removed did not impact performance when resource allocation was decreased. Based on this result of this gradient, AutoTune would identifies following MRs as being IMRs: (Node, cpu), (Mongo, memory), and (Mongo, cpu). When compared to the the actual performance improvements for the same MRs (Figure 9), we see that provisioning more resources to those exact three MRs results in performance gains. The other MRs – correctly identified as NIMRs – did not result in significant performance improvements when the resource allocation was increased. For the MEAN stack, the stressing mechanism positively identified all the IMRs. Additionally, note that provisioning more resources to the most impacted MR causes end-to-end performance to improve by 752 ms (which is nearly 30%). It is important to re-emphasize here that the Performance Improvement Phase makes no guarantees about the exact *magnitude* of the performance improvement upon the mitigation of the bottleneck.

We observed similar behavior among the Hotrod and Apartment Apps.

6.4.3 Performance Improvement Phase: Resource Transfer Example

Now that we have shown that AutoTune can positively identify IMRs, we discuss concretely how resource transfer works. Recall that the primary way the gradient phase improves performance is by transferring resources between less impacted MRs and more impacted MRs. Transferring resources in this way should work well if everything was cleanly linear in behavior. But systems are not so nicely behaved, and simultaneously taking two actions (removing resources from an NIMR and giving them to an IMR) could yield unexpected behavior.

To evaluate how well this works in practice, we observed the ability of AutoTune to transfer resources in various setups. Within the Apartment App, we zoomed into the ELK stack, which consists of three microservices: Elasticsearch, Logstash, and Kibana. CPU-Quota was initially evenly distributed between Elasticsearch and Kibana. The baseline performance under this configuration was measured to be 31.49 seconds. In the first iteration, AutoTune identifies Elasticsearch CPU-Quota as the most impacted MR. Upon reducing the resource allocation of $(Kibana, cpu-quota)$ by 25% and increasing $(Elasticsearch, cpu-quota)$ by the same amount, performance improved to 28.48 seconds, equivalent to the performance exhibited simply by improving $(Elasticsearch, cpu-quota)$ by 25% whilst leaving $(Kibana, cpu-quota)$ to be constant. We observed similar behavior for the MEAN stack.

While this does not cover the space of all application behaviors (even within the applications that we explored), thus far we have not seen an application where transfering resources from a correctly identified non-impacted MR and provisioning it to a correctly identified most-impacted MR has resulted in undesirable behavior. In the event of this undesirable behavior, AutoTune is capable of detecting it. Once detected, the *backtrack step* would simply take over and ultimately undo the resource transfer.

6.5 Additional Issue: Effect of Interference

In this section, we dive into the issue of interference, which we discussed in Section 3 (Additional Issues). In particular, we illustrate how AutoTune copes with interference on the HotRod application, the only application we tested where placement contributed to the application performance. This is because the HotRod applications exhibits a significant number of compute-impacted MRs. AutoTune empirically detects interference by measuring the clamped down performance with the original placement and compares it with the performance on the new compacted placement. For the HotRod application, performance degraded by 12% – despite having the same resource allocations.

Recall that AutoTune reduces the likelihood of new interference by 1.) preserving colocated microservice instances from the initial deployment and 2.) exclusively assigning core(s) to those microservices. Upon applying affinity-based placements and core pinning, performance degradation upon clampdown decreased to 2%. With this mechanism, more

[2]Note that we can do this kind of resource increase for this special test, but when running AutoTune such increases are not possible because they typically violate the resource constraints on the server.

compute resources were utilized, but the number of packed nodes remained constant. Ultimately, for this version of the HotRod application, this simple mechanism improves performance after the clampdown phase at no additional cost to the operator. There are clear limitations to this approach, especially when all compute MRs are *similarly* impacted. Our solution here does not attempt to exhaustively eliminate container interference; other projects have addressed this problem extensively [27] [41]. Nevertheless, in the only case where our applications encountered interference, the combination of core pinning and invariant placement sufficiently mitigated the effect of placement on performance.

7 Discussion

Software Configurations as AutoTune Inputs: Some services, *e.g.,* Spark, require that configuration be updated (*e.g.,* by increasing heap size) to take advantage of additional resources. The complexity of these configuration changes varies across services: some services such as Nginx allow configuration to be updated at runtime, while others such as Spark need to be restarted before they can make use of additional resources. The user is required to provide this configuration as input. This requirement to change configuration adds noticeable overheads when getting started with and deploying AutoTune. As a result AutoTune is better suited to handling services that are designed to dynamically adapt to using additional resources at runtime. In our own experiences running AutoTune, most reconfiguration programs are expressible as a simple Bash script. We however note that several services (e.g., Postgres [17], NGINX) already provide tools (*e.g.,* `pgtune`) and APIs or dynamic configuration. Furthermore, tools such as Kubernetes Helm [1] have begun developing tools that apply across services. We plan to more fully investigate the challenges of integrating with automatic configuration tools in future work.

Representative Workload: Prior work [25,43] has observed that application requirements can vary widely depending on workloads, and thus it is important that operators use realistic workloads. How should representative workload traces be generated? In this work we assume that operators provide us with realistic workloads, which might be generated by recording production traces, through the use of tools such as Kraken [43], through the use of benchmarks developed by organizations such as TCP [42], SPEC [36], etc. Furthermore, we note that while interesting, the problem of generating representative traces is orthogonal to the resource allocation problem we address.

8 Related Work

Recent work to determine resource allocations has largely focused on provisioning resources for *individual microservices*, rather than considering complex dependencies in end-to-end heterogeneous microservice applications. These projects can take either a data-driven or domain specific approach. Examples of work that adopt the data-driven approach include Paragon [12] and Quasar [11] that rely on microarchitectural details to reduce colocation overheds. While these works do not make any assumptions about the application, and can account for behaviours such as noisy neighbors (which are ignored by AutoTune), they cannot scale to applications containing several microservices. Domain specific approaches include work such as Ernest [44], CherryPick [5] and Paris [45] which make strong assumptions about application structure (*e.g.,* assuming map-reduce like data parallel frameworks) or that the application is comprised of homogeneous services. Other approaches to the resource allocation problem have relied on the use of instrumentation and custom tooling to infer application dependencies. This approach adds performance overheads and increases system complexity complexity. Sieve [38] considers performance across multiple microservices, but requires instrumentation like *sysdig* or *dtrace*, with overhead of up to 100% in the worst case [16]. Additionally, these tools suffer from causation ambiguity and leads to high rates of false positives [24,26]. Other proposed systems allow operators to infer performance behaviors of various systems, but they largely require significant modifications to the hypervisor or to the application under test [19,20] or only explore performance improvements along a single resource dimension (e.g., network) [6,20,30]. Other works have suggested improving application performance through profiling programs and optimizing code [8,9,29]; these provide a different set of knobs from impacted MR allocations, and thus can be jointly used with AutoTune to improve application performance. Past proposals on resource scheduling [14,15,28] assume that the administrator provides resource requirements as input; thus we view this work as complimentary to AutoTune.

9 Conclusion

It is important to consider AutoTune relative to its goals. AutoTune was not intended to provide optimal performance, nor optimal efficiency, nor provable guarantees, nor to work well with arbitrarily varying workloads. Instead, it was designed to be easy-to-use (operators need only provide an initial deployment, a representative workload, and a performance metric) and generally applicable (it does not make any assumptions about the nature of the application) tool that reduces the number of servers needed to deploy applications while maintaining (or improving) performance. We believe AutoTune achieves this goal, and we are not aware of any other tool that does so. However, only widespread use of AutoTune will allow us to fully understand its limitations, which hopefully will lead to further improvements.

References

[1] Helm – the kubernetes package manager. https://helm.sh/, retrieved 9/17/2018.

[2] Horizontal pod autoscaler. https://kubernetes.io/docs/tasks/run-application/horizontal-pod-autoscale/, retrieved 9/16/2019.

[3] Placing pods relative to other pods using affinity and anti-affinity rules. https://docs.openshift.com/container-platform/4.1/nodes/scheduling/nodes-scheduler-pod-affinity.html, retrieved 9/16/2019.

[4] ab - Apache HTTP server benchmarking tool. https://httpd.apache.org/docs/2.4/programs/ab.html, retrieved 10/27/2017.

[5] O. Alipourfard, H. H. Liu, J. Chen, S. Venkataraman, M. Yu, and M. Zhang. Cherrypick: Adaptively unearthing the best cloud configurations for big data analytics. In *NSDI*, 2017.

[6] J. S. Bucy, J. Schindler, S. W. Schlosser, and G. R. Ganger. *The DiskSim Simulation Environment Version 4.0 Reference Manual*. CMU PDL, 2008.

[7] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes. Borg, omega, and kubernetes. *Commun. ACM*, 59:50–57, 2016.

[8] M. Burtscher, B.-D. Kim, J. R. Diamond, J. D. McCalpin, L. Koesterke, and J. C. Browne. Perfexpert: An easy-to-use performance diagnosis tool for hpc applications. In *SC*, 2010.

[9] C. Curtsinger and E. D. Berger. Coz: Finding code that counts with causal profiling. In *SOSP*, 2015.

[10] J. Dean and L. A. Barroso. The tail at scale. *Commun. ACM*, 56:74–80, 2013.

[11] C. Delimitrou and C. Kozyrakis. Quasar: Resource-efficient and qos-aware cluster management. In *Proceedings of the 19th International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '14, pages 127–144, New York, NY, USA, 2014. ACM.

[12] C. Delimitrou and C. E. Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. In *ASPLOS*, 2013.

[13] Docker Inc. What is a Container? https://www.docker.com/what-container retrieved 08/03/2017.

[14] A. Ghodsi, M. Zaharia, B. Hindman, A. Konwinski, S. Shenker, and I. Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *NSDIR*, 2011.

[15] R. Grandl, G. Ananthanarayanan, S. Kandula, S. Rao, and A. Akella. Multi-resource packing for cluster schedulers. In *SIGCOMM*, 2014.

[16] B. Gregg. dtrace pid provider overhead. http://dtrace.org/blogs/brendan/2011/02/18/dtrace-pid-provider-overhead/ retrieved 05/01/2018, 2011.

[17] B. Gregg. Tuning Your PostgreSQL Server. https://wiki.postgresql.org/wiki/Tuning_Your_PostgreSQL_Server retrieved 05/01/2018, 2011.

[18] A. Gupta. Microservice design patterns. http://blog.arungupta.me/microservice-design-patterns/, retrieved 9/17/2018.

[19] D. Gupta, K. V. Vishwanath, M. McNett, A. Vahdat, K. Yocum, A. Snoeren, and G. M. Voelker. Diecast: Testing distributed systems with an accurate scale model. *ACM Transactions on Computer Systems*, 29(2):4:1–4:48, May 2011.

[20] D. Gupta, K. Yocum, M. McNett, A. C. Snoeren, A. Vahdat, and G. M. Voelker. To infinity and beyond: Time warped network emulation. In *SOSP*, 2005.

[21] A. A. Hagberg, D. A. Schult, and P. J. Swart. Exploring network structure , dynamics , and function using networkx. In *SciPy*, 2008.

[22] B. Hubert. tc(8). Linux man page – iproute2, 2001.

[23] E. Jackson. Kelda: An approachable way to deploy the cloud. https://github.com/kelda/kelda, retrieved 9/17/2018.

[24] Y. Ji, S. Lee, E. Downing, W. Wang, M. Fazzini, T. Kim, A. Orso, and W. Lee. Rain: Refinable attack investigation with on-demand inter-process information flow tracking. In *CCS*, 2017.

[25] S. A. Jyothi, C. Curino, I. Menache, S. M. Narayanamurthy, A. Tumanov, J. Yaniv, R. Mavlyutov, I. Goiri, S. Krishnan, J. Kulkarni, and S. Rao. Morpheus: Towards automated slos for enterprise clusters. In *OSDI*, 2016.

[26] K. H. Lee, X. Zhang, and D. Xu. High accuracy attack provenance via binary-based execution partition. In *NDSS*, 2013.

[27] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis. Heracles: Improving resource efficiency at scale. In *Proceedings of the 42Nd Annual International Symposium on Computer Architecture*, ISCA '15, pages 450–462, New York, NY, USA, 2015. ACM.

[28] H. Mao, M. Alizadeh, I. Menache, and S. Kandula. Resource management with deep reinforcement learning. In *HotNets*, 2016.

[29] B. P. Miller, M. D. Callaghan, J. M. Cargille, J. K. Hollingsworth, R. B. Irvin, K. L. Karavanic, K. Kunchithapadam, and T. Newhall. The paradyn parallel performance measurement tool. *IEEE Computer*, 28:37–46, 1995.

[30] R. Pan, B. Prabhakar, K. Psounis, and D. Wischik. Shrink: a method for enabling scaleable performance prediction and efficient network simulation. *IEEE/ACM Transactions on Networking*, 13:975–988, 2005.

[31] pgtune(8): PostgreSQL COnfiguration Tuner. https://linux.die.net/man/8/pgtune retrieved 08/03/2017.

[32] Redhat Resource Management Guide. https://goo.gl/hiFDD7.

[33] D. K. Rensin. *Kubernetes - Scheduling the Future at Cloud Scale*. O'Reilly Media, 2015.

[34] C. Richardson. Pattern: Microservice architecture. https://microservices.io/patterns/microservices.html, retrieved 9/17/2018.

[35] Y. Shkuro. Take opentracing for a hotrod ride. https://medium.com/opentracing/take-opentracing-for-a-hotrod-ride-f6e3141f7941, retrieved 9/17/2018.

[36] Standard Performance Evaluation Corporation. http://spec.org/, retrieves 10/27/2017.

[37] L. Suresh, J. Loff, F. Kalim, N. Narodytska, L. Ryzhyk, S. Gamage, B. Oki, Z. Lokhandwala, M. Hira, and M. Sagiv. Automating Cluster Management with Weave. *arXiv e-prints*, page arXiv:1909.03130, Sep 2019.

[38] J. Thalheim, A. Rodrigues, I. E. Akkus, P. Bhatotia, R. Chen, B. Viswanath, L. Jiao, and C. Fetzer. Sieve: Actionable insights from monitored metrics in microservices. *CoRR*, abs/1709.06686, 2017.

[39] Todd Hoff. Lessons Learned From Scaling Uber To 2000 Engineers, 1000 Services, And 8000 Git Repositories. https://goo.gl/1MRvoT, retrieved 01/21/2017.

[40] Tony Mauro. Adopting Microservices at Netflix: Lessons for Architectural Design. https://goo.gl/DyrtvI, retrieved 01/21/2017.

[41] A. Tootoonchian, A. Panda, C. Lan, M. Walls, K. Argyraki, S. Ratnasamy, and S. Shenker. Resq: Enabling slos in network function virtualization. In *Proceedings of the 15th USENIX Conference on Networked Systems Design and Implementation*, NSDI'18, pages 283–297, Berkeley, CA, USA, 2018. USENIX Association.

[42] Transaction Processing Council. http://www.tpc.org/, retrieved 10/27/2017.

[43] K. Veeraraghavan, J. Meza, D. Chou, W. Kim, S. Margulis, S. Michelson, R. Nishtala, D. Obenshain, D. Perelman, and Y. J. Song. Kraken: Leveraging live traffic tests to identify and resolve resource utilization bottlenecks in large scale web services. In *OSDI*, 2016.

[44] S. Venkataraman, Z. Yang, M. Franklin, B. Recht, and I. Stoica. Ernest: Efficient performance prediction for large-scale advanced analytics. In *NSDI*, 2016.

[45] N. J. Yadwadkar, B. Hariharan, J. E. Gonzalez, B. Smith, and R. Katz. Selecting the best vm across multiple public clouds: A data-driven performance modeling approach. In *SOCC*, 2016.