ARTIFACT EVALUATED usenix ASSOCIATION AVAILABLE
ARTIFACT EVALUATED usenix ASSOCIATION FUNCTIONAL

Cilantro: Performance-Aware Resource Allocation for General Objectives via Online Feedback

Romil Bhardwaj*[1], Kirthevasan Kandasamy*[2], Asim Biswal[1], Wenshuo Guo[1], Benjamin Hindman[1], Joseph Gonzalez[1], Michael Jordan[1], and Ion Stoica[1]

[1]UC Berkeley
[2]University of Wisconsin-Madison

Abstract

Traditional systems for allocating finite cluster resources among competing jobs have either aimed at providing fairness, relied on users to specify their resource requirements, or have estimated these requirements via surrogate metrics (e.g. CPU utilization). These approaches do not account for a job's real world performance (e.g. P95 latency). Existing performance-aware systems use offline profiled data and/or are designed for specific allocation objectives. In this work, we argue that resource allocation systems should directly account for real-world performance and the varied allocation objectives of users. In this pursuit, we build Cilantro.

At the core of Cilantro is an online learning mechanism which forms feedback loops with the jobs to estimate the resource to performance mappings and load shifts. This relieves users from the onerous task of job profiling and collects reliable real-time feedback. This is then used to achieve a variety of user-specified scheduling objectives. Cilantro handles the uncertainty in the learned models by adapting the underlying policy to work with confidence bounds. We demonstrate this in two settings. First, in a multi-tenant 1000 CPU cluster with 20 independent jobs, three of Cilantro's policies outperform 9 other baselines on three different performance-aware scheduling objectives, improving user utilities by up to $1.2-3.7\times$ and performs comparably to oracular policies. Second, in a microservices setting, where 160 CPUs must be distributed between 19 inter-dependent microservices, Cilantro outperforms 3 other baselines, reducing the end-to-end P99 latency to $\times 0.57$ the next best baseline.

1 Introduction

The goal of cluster resource managers is to allocate a finite amount of scarce resources to competing jobs. When doing so, we should ensure that the allocations fulfill the users' and the organization's overall goals. Traditionally, resource allocation policies have aimed to provide fairness [16, 24], maximize resource utilization [61], maximize the amount of work done [24], or minimize queue lengths [47, 66]. However, these policies miss, or at best are imperfect proxies for what matters most to the users: the performance of their jobs in terms of real-world metrics that impact business (e.g. P99 latency or throughput for a serving job). Barring some recent exceptions [10, 18, 35, 64], resource allocation systems have traditionally focused on the resources requested by a job rather than the job's real-world performance from using those resources (henceforth, simply *performance*).

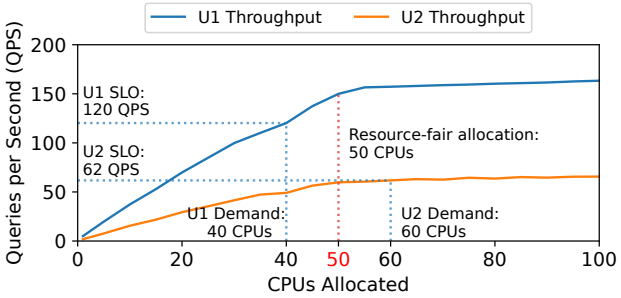


Figure 1: Two users, U1 and U2, serving TPC-DS benchmark queries with different resource-throughput mappings and performance goals (SLO). A user's demand is the amount of CPUs needed for her SLO.

To illustrate the pitfalls of performance-oblivious scheduling, consider an example where two users, U1 and U2, are sharing a cluster of 100 CPUs. They are each serving different sets of TPC-DS [43] queries and care about their throughput: U1's service level objective (SLO) is 120 queries per second (QPS), while the U2's SLO is 62 QPS. If the goal is to satisfy all user's SLOs, how should CPUs be allocated? If it were known that the resource-to-throughput curves of the two users' jobs were as shown in Figure 1, a scheduler can allocate 40 CPUs to the first job and 60 to the second. However, in practice, this mapping is usually not available and performance-oblivious scheduler will likely be suboptimal. For instance, a CPU-based fair allocation algorithm would allocate 50 CPUs to each user, which would result in U2 getting just 59 QPS, thus missing its SLO.

* Co-primary authors.

Despite extensive theoretical work [16, 24, 28, 37, 38], performance-aware scheduling has remained challenging since the *resource-to-performance mappings* are usually unavailable in practice. To obtain these mappings, past work [15, 58, 64] profile their workloads before execution. Such profiling has three limitations. First, offline profiled resource-to-performance mappings may not reliably reflect a job's performance in a production environment, as it may not capture the interference from other jobs [14] and the server's performance variability [19]. Second, jobs' resource requirements change with time due to varying load (e.g., arrival rate of external queries) and profiling typically cannot account for these changes. Third, such profiling is burdensome for users and expensive for organizations as it requires a large pool of resources to exhaustively profile a wide range of resource allocations. This informs the *first requirement* for this work: obtain the resource-to-performance mappings in the production environment where the job will be run.

Even if the resource-to-performance mappings are known, the choice of scheduling policy depends on the objective of the end-users (e.g. organization, developers). For instance, suppose in Figure 1, we wished to maximize the total throughput of the cluster, instead of trying to satisfy each user's SLOs. In this case, we would allocate ∼64 CPUs to U1 and ∼36 to U2 for a total throughput of ∼212 QPS. As more realistic examples, in multi-tenant clusters, we may wish to use policies which balance between performance and fairness [16, 24, 38]. In contrast, when we provision resources to different microservices of the same application, we are more interested in some end-to-end performance objective, such as application latency, and may wish to allocate more resources to critical microservices which bottleneck performance. These objectives can vary from organization to organization and optimizing for such different objectives requires different allocation policies. However, while end users may find it relatively easy to state their objective (e.g., satisfy all SLOs, maximize throughput), it is harder to design a policy to achieve it. This informs our *second requirement*: support a diverse set of user-defined scheduling objectives.

To address these requirements, we introduce Cilantro, a framework for performance-aware allocation of a single fungible resource type (e.g. CPUs, containers) among competing jobs (Figure 2). In Cilantro, end users first declare their desired scheduling objective. To satisfy the first requirement, a pool of performance learners and load forecasters analyzes live feedback from jobs and learns models to estimate resource-performance curves and load shifts for each job. To satisfy the second requirement, Cilantro's scheduling policies, which are automatically derived based on the users' objectives, leverage these estimated models to compute allocations for each job. As the learned models become accurate over time, Cilantro is able to eventually achieve the users' objectives. This obviates the need for an offline model to estimate the required

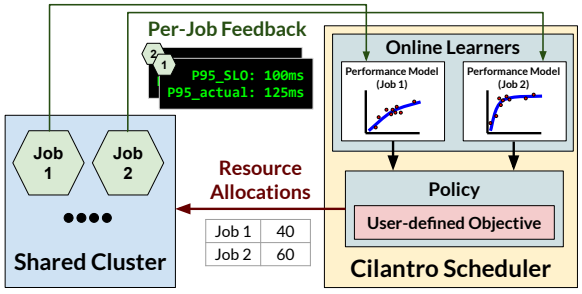


Figure 2: Cilantro overview. Cilantro uses continuous feedback to dynamically learn each job's resource-to-performance mappings. An uncertainty-aware resource allocation policy, instantiated for the user's objective, uses these mappings to determine allocations.

resource allocation for a given performance target, and allows Cilantro to optimize for custom objectives, such as various fairness or performance criteria. This is a marked departure from performance-oblivious policies, those based on unreliable proxy metrics such as CPU utilization and queue lengths, and other heuristic-based policies (using either surrogates [51] or performance metrics [10, 18]) which are designed for very specific scheduling objectives. Cilantro seamlessly enables the implementation of performance-aware policies in two settings: *(i)* multi-tenant resource allocation for independent jobs, and *(ii)* resource allocation for inter-dependent jobs (microservices) within an application.

Our proposed solution solves two key challenges. First, estimating resource-to-performance mappings online can be notoriously difficult due to highly stochastic nature of real-time production environments, unexpected load shifts, especially in the early stages when there is insufficient data. To operate without accurate estimates, Cilantro informs scheduling policies with confidence intervals of its estimates. Policies are designed to account for this uncertainty when making allocation decisions until the estimates become more accurate. Accounting for this uncertainty helps Cilantro conservatively explore the space of allocations making it robust to environment stochasticity and also to the idiosyncrasies specific to the performance models used.

Second, supporting a diversity of objectives in the same framework is challenging. The monolithic design of end-to-end feedback-driven approaches [34,49,64] restricts them only the objective they were originally designed for. Instead, Cilantro achieves generality in supporting custom objectives by decoupling the learning mechanisms from the allocation policy. This decoupling is necessary as it allows us to account for the effect of each job's performance and load shifts on the objective individually. Moreover, this decoupling has other intangible benefits: it leads to a more transparent design which is easy to debug than monolithic systems which directly optimize for end-to-end performance, and if online job feedback cannot be obtained for a particular job, it is easy to swap the learners with profiled information or other sensible defaults.

We have implemented Cilantro as an open-source extension to the Kubernetes core scheduler, available at https://github.com/romilbhardwaj/cilantro. To evaluate Cilantro, first we deploy it on a 1000-CPU multi-tenant cluster which includes a diversity of real-world, latency and throughput-sensitive jobs. On three different allocation objectives, Cilantro's policies are able to outperform 9 other baselines, and is able to compete with oracular policies which know the resource-to-performance mappings a priori on resource efficiency and fairness. When compared to resource-fair allocation, it is able to increase the performance of $1/3$ of users in the clusters by $1.2-3.7\times$. Second, we evaluate Cilantro on a 160 CPU cluster where we wish to allocate CPUs to constituent microservices of an application. Here, Cilantro is able to minimize the end-to-end P99 latency of the application to $\times 0.18$ the latency of a resource-fair scheduler and to $\times 0.57$ of the next-best performance-aware baseline.

2 Background & Related Work

In this section, we compare Cilantro with prior work. Table 1 summarizes the key differences of Cilantro against other resource allocation systems and methods.

Performance oblivious methods: The simplest, yet popular approach to allocating finite resources among competing jobs, is to adopt a *resource fair* policy, which simply divides the resource equally (or proportional to weights) [2, 30, 32, 42]. As this does not account for jobs' resource requirements, it is inadequate in all but the most trivial settings.

Several scheduling frameworks, such as Kubernetes [9], Mesos [29] and YARN [57], relies on users to submit their own resource demand. To execute resource allocations from policies, Kubernetes and YARN use resource reservations while Mesos negotiates through resource offers. This requires users to estimate their jobs' resource needs, which can be difficult. They focus on one-way resource allocations and do not provide any mechanisms for the policy to get feedback on application performance. However, recognizing that end users may have varied scheduling objectives, these frameworks support and implement multiple policies.

Methods based on proxy metrics: The most common approach to account for resource requirements relies on proxy metrics (e.g. CPU utilization, work-queue lengths). Quasar [15] offline profiles jobs' proxy metrics, and has a fixed operator-centric policy to maximize cluster utilization. Paragon [14] accounts for resource heterogeneity and inter-job interference to achieve performance guarantees. AGILE [46] models the resource pressure, and uses demand prediction to minimize SLO violations. The above works do not directly account for users' performance goals and optimize for singular objectives.

Methods which use offline profiling: Some work has explored directly incorporating job performance via profiled historical data. Morpheus [33] aims to mitigate performance unpredictability by defining SLOs and satisfying their resource demands by using models based on historical data. Ernest [58] provides methods for estimating performance curves using limited amount of data, but does not study using these estimates for resource allocation under scarcity. Sinan [64] partly uses profiled information for auto-scaling in a cloud environment. Quincy's [30] min-cost flow formulation aims at providing fairness, but relies on offline estimates of data movement costs. For reasons explained in §1, offline profiling can be problematic and it is desirable to rely on real-time feedback to determine resource allocations.

Methods which use online feedback: Among related work, some feedback-driven systems account for performance metrics and SLOs in resource allocation. Jockey [18] focuses on meeting latency SLOs for a single job by modeling internal job dependencies to dynamically re-provision resources. Henge [35] defines new utility functions for stream processing workloads and aims to maximize a singular objective – the sum of utility of all jobs. [48] uses application hints in for prefetching disk blocks in the OS kernel. Gavel [44] is a scheduler for ML training workloads in heterogeneous environments with varying objectives. Since Gavel is focused on ML training, it's policies are designed for throughput and a greedy optimizer computes the optimal allocation for each round. On the other hand, Cilantro supports any metric specified by the user and employs online learning to eventually converge on the optimal allocation. Finally, in a video streaming application, Minerva [45] studies methods for resource allocation so that all end users have the same quality of service. The highly customized policies used in the above works, while adequate to the allocation objectives set out by the authors, are not applicable for diverse cluster objectives which is our goal here.

Variable resource amounts: In other related work, PARTIES [10] allocates resources to jobs within the same server while always satisfying SLOs. If the SLOs of all jobs cannot be met, it evicts one of them to a different server; thus, it does not apply to our setting where there is a fixed amount of resources and eviction is not possible. Indeed, in §7 we show that a straightforward adaptation of PARTIES does not work as well. Sinan [64], DS2 [34], Autopilot [51] and FIRM [49] consider performance-aware resource allocation using online feedback when there is elasticity in resource availability, e.g. the cloud. Because these works can scale up to more resources than originally provisioned, they are not directly comparable to Cilantro which operates in a fixed cluster setting. While the cloud is an emerging use case, traditional fixed resource cluster management remains pertinent for privacy and cost reasons. Moreover, the above work focus on specific goals and are not designed to handle general allocation objectives. As an example, FIRM [49] focuses on autoscaling resources for single applications deployed as microservices to

	Cilantro	PARTIES [10]	Henge [35]	Autopilot [51]	Jockey [18]	Paragon [14]	Morpheus [33]	DS2 [34]	Quasar [15]	FIRM [49]	Sinan [64]	YARN [57]	Mesos [29]
Performance awareness	RW	RW	RW	RW	RW	RW	RW	RW	PM	PM	PM	PO	PO
Works without apriori performance model?	Y	Y	Y	Y	N	N	N	N	Y	Y	N	NA	NA
Supports multiple allocation objectives?	Y	N	N	N	N	N	N	N	N	N	N	Y[1]	Y[1]
Cluster size	Fix	Var	Fix	Var	Fix	Fix	Fix	Var	Fix	Var	Var	Fix	Fix

abbr. RW = Real-world metrics, e.g., latency, PM = Proxy metrics e.g., CPU util., PO = Performance oblivious, Fix = Fixed size, Var = Variable size
[1] Supports multiple objectives, but only performance oblivious ones

Table 1: Cilantro and related work. Cilantro uses real-world metrics (e.g., latency) to build performance models online, which can be used to derive custom policies for different objectives.

minimize end-to-end SLO violations, Cilantro operates differently, reallocating a fixed number of resources according to user-specified objectives, which can include fairness considerations. Additionally, FIRM uses Reinforcement Learning with anomaly injection, in contrast to Cilantro, which focuses on resource-allocation under uncertainty and is agnostic to the learning method used.

3 Cilantro Architecture

Cilantro is a performance-aware scheduling framework that can optimize for various scheduling objectives without requiring any a priori knowledge of the resource-performance mapping of the workloads. The design of Cilantro is informed by the following two key insights.

[I1] Offline profiling of resource-performance is insufficient. Performance-aware policies rely on accurate estimates of resource-to-performance mappings and load shifts. Offline profiling of these resource-performance mappings can be inaccurate due to unpredictability in server and application performance [19] and changing traffic patterns [50]. Adapting to these changes necessitates continuously learning and predicting these unknowns in an online manner.

[I2] Decoupling learning mechanisms and policies enables diverse scheduling objectives. As different scheduling policies optimize different criteria, it may be challenging for a scheduling framework to generally support different policy types. Prior work on feedback-driven resource allocation [34, 49, 64] uses an end-to-end model for allocating resources for a fixed objective, such as total utility or cost. Optimizing for a different objective in these systems may require a complete redesign of the system and policy, or at the very least an expensive retraining of their models. Decoupling learning mechanisms from policies allows the model to be learned once and applied to multiple allocation objectives. This decoupling also increases transparency in the allocation decisions made by the scheduler and facilitates debugging.

We leverage these learnings to build Cilantro (Figure 3). Cilantro is composed of two key components: the centralized Cilantro scheduler, which is responsible for generating resource allocations, and the Cilantro clients—lightweight sidecars co-located with each job—which fetch a job's performance metrics and send them to the Cilantro scheduler. Informed by **[I1]**, the Cilantro scheduler employs online learning to create increasingly accurate models of job performance and load. Guided by **[I2]**, the policy optimizes a user-defined objective by polling these models for a resource-performance estimates to produce a resource allocation.

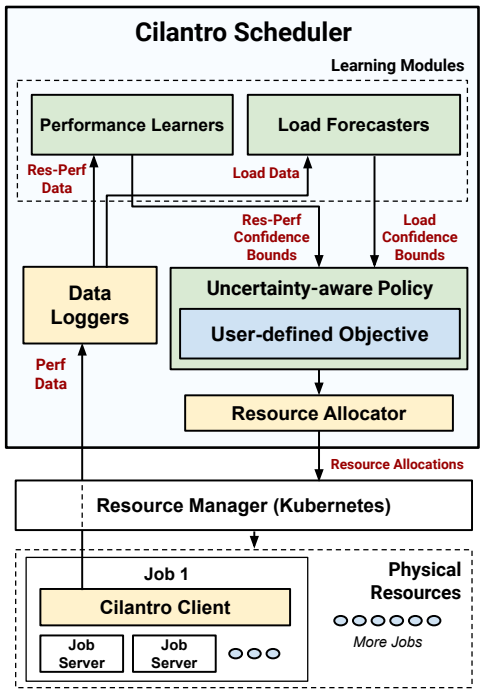


Figure 3: The Cilantro scheduler and client architecture. The scheduler generates resource allocations for jobs and the clients collect performance feedback to report to the scheduler.

Assumptions & terminology. In this work, we will focus on jobs which can scale elastically with the number of resources with corresponding gains in performance. Examples of such workloads include stateless or stateful distributed

services (e.g., prediction serving [13], memcached [21], Cassandra [40]), distributed computation (ML training, MPI jobs) and distributed frameworks (e.g. Hadoop [52], Spark [62]). Some of these can be viewed as a collection of several tasks whose job size may vary with time, such as in serving jobs. Each task may refer to a query whose arrival rate may change with time. For jobs with such varying query rates, we will refer to the instantaneous rate of external query arrival as the *load* (measured in queries per second (QPS)). Finally, we assume there is a *fixed amount* of a *single*, *fungible* resource type that must be allocated.

Cilantro scheduler: The Cilantro scheduler is designed as a centralized asynchronous event driven system. Event sources include timers, performance updates received from the Cilantro clients, and cluster state updates from the underlying resource manager. Below, we describe the scheduler's modules. Specific implementation details are available in §6.

1. Data loggers. Application metrics pushed from Cilantro clients are stored in memory-backed tables. They relay these metrics to the performance learners and load forecasters.

2. Performance learner. The performance learner learns a job's performance as a function of the resource allocation and the load using an associated model. It periodically polls the data logger for new data and updates the model. The learner's update frequency is constrained only by the velocity at which the model can be updated. One instance of a performance learner is maintained per application. A performance learner provides `get-perf-ucb` and `get-perf-lcb` interfaces for a policy, which return upper and lower confidence bounds for the performance as a function of the resources and load.

3. Load forecasters. In many real-world deployments, the job size could vary with time depending on the real-time traffic, which should be accounted for when allocating resources. The goal of the load forecaster is to estimate this load for the duration of a future allocation based on past observed loads via an associated time series model. It offers `get-load-ucb` interface for a policy which returns an upper confidence bound for the future load. Load forecasters are periodically updated by polling from the data loggers.

4. Uncertainty-aware Policy. Policies compute allocations in order to optimize for a user-specified scheduling objective. In an online setting, using direct estimates of the performance may fail as it does not reflect the uncertainty in the model. Therefore, Cilantro's policies leverage confidence intervals of these estimates to account for this uncertainty in a principled manner when making allocation decisions (§4).

5. Resource allocator. The resource allocator is responsible for executing the resource allocations by interfacing with the underlying cluster manager. This module is driven via an allocation expiry event, upon which it invokes the policy's `compute-alloc` method and allocates the resources. Allocation expiry events are raised based on a timeout, resulting in a new round of allocations. In practice, the duration of an allocation round is limited by the agility of the environment. Since scaling jobs requires time, changing resource allocations too frequently can result in job thrashing (having to scale down before it has a chance to utilize new resources).

Cilantro client: The Cilantro client is a lightweight side-car container whose purpose is to to poll the job to get its current performance, process it, and publish it to the scheduler's data loggers. The primary task for the client is to extract metrics from their assigned job. Many systems expose REST endpoints to query system performance [3, 4], but often the applications also use monitoring tools such as Prometheus or Grafana. Depending on the job, the performance metric extraction logic is specified by the users. In §5, we describe built-in fallback options if job metrics are not available.

4 Policies

We now describe our policies for performance-aware resource allocation in two settings: multi-tenant resource allocation in a fixed cluster (§4.1), and allocating finite resources to constituent microservices of an application (§4.2).

Set up & notation: We will denote the number of jobs (or microservices) by n, the amount of resources by R, and an allocation by $a = (a_1, \ldots, a_n)$, where a_j is the amount of resources allocated to job (or microservice) j. A scheduler should allocate these resources so that $\sum_{j=1}^{n} a_j \leq R$.

4.1 Resource allocation in shared clusters

Cilantro supports two classes of performance-aware allocation objectives in the multi-tenant setting: welfare-based, and demand-based. Our primary contributions are in §4.1.2 where we derive uncertainty-aware online variants of these policy classes. But first, we will review some common examples of such objectives in §4.1.1. For what follows, we will need to define the *performance*, *demand*, and *utility* of a job.

Performance: The *resource/load-to-performance mapping* (henceforth simply performance or performance mapping) p_j of a user's job j refers to some raw metric of interest, which, say, can be obtained from a monitoring tool. We write the performance $p_j(a_j, \ell_j)$ as a function of the resources received a_j and the load ℓ_j. As we are allocating a single resource type, a_j is a single number, as is ℓ_j. For example, in a serving job with a P95, 100 ms latency SLO, the performance may be the fraction of queries completed in under 100 ms, and the load may refer to the external arrival rate of queries.

Demand: If a job has a well-defined SLO, we define the *demand* d_j to be the minimum amount of resources needed to achieve this SLO. The demand depends on the job's performance curve p_j, SLO, and load ℓ_j.

Utility: The *utility* u_j of a job is the *practical value* derived

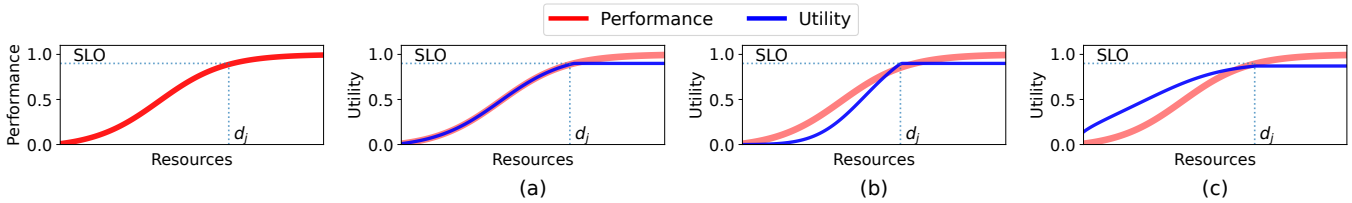


Figure 4: Three candidates for SLO-based utility functions. The left-most figure shows a job's performance p_j as a function of the resources (for fixed load). In (a), the utility scales linearly with performance until the SLO, i.e $u'_j(p) \propto \min(p, \text{SLO})$, whereas in (b) it scales quadratically $u'_j(p) \propto \min(p, \text{SLO})^2$, and in (c) it scales with the square-root $u'_j(p) \propto \min(p, \text{SLO})^{1/2}$. Here, (b) captures settings where even small SLO violations are critical while (c) captures settings where small SLO violations are not very significant.

due to its performance. Generally, u_j is a non-decreasing function of the performance and we can write $u_j(a_j, \ell_j) = u'_i(p_j(a_j, \ell_j))$ for some non-decreasing function u'_j.

Examples of utilities. The simplest option is to set the utility to be equal to the performance $u_j = p_j$, i.e., u'_j is the identity. However, we may also choose a utility which is more applicable when there are well-defined SLOs. Fig. 4 illustrates three candidates for u'_j: the maximum utility for any job is set to 1, which is achieved for any performance greater than the SLO; for performances below the SLO, we may set the utility to *(a)* decrease proportionally with SLO violation, *(b)* decrease sharply in settings where small SLO violations are critical (e.g., with external customers where SLO violations can lead to penalties [1] and a loss of credibility), *(c)* decrease gradually when small SLO violations are not critical (e.g., soft SLOs internal to an organization). Such utility forms which are 'clipped' at the SLO provide a simple way to compare jobs with heterogeneous performance metrics and SLOs, such as latency and throughput. Prior work have also used similar forms of utility [23,35,60]. For these reasons, our experiments also use these forms, although we emphasize that Cilantro can handle any utility form which increases with performance.

4.1.1 Review of multi-tenant allocation when performance mappings are known

We will first review two classes of multi-tenant allocation objectives supported in Cilantro—welfare-based and demand-based—and three examples of such objectives. In §4.1.2, we will develop online learning policies that achieve the same objectives when performance mappings are unknown.

Welfare-based objectives: These policies aim to maximize a given cluster-wide *welfare* function W, which is a function of the utility of each job, i.e., $W = W(u_1, \ldots, u_n)$. Below, we describe two common welfare-based objectives.

(i) Social welfare (a.k.a. Kelly mechanism [38]): We choose the allocation a which maximizes the social welfare (the average utility), i.e. $a = \text{argmax}\, W_\text{S}$, where,

$$W_\text{S} = \frac{1}{n}\sum_{j=1}^{n} u_j(a_j, \ell_j) = \frac{1}{n}\sum_{j=1}^{n} u'_j(p_j(a_j, \ell_j)). \quad (1)$$

As we show in Figure 5, this notion of fairness allocates more resources to "high-performing" users, i.e those who can generate large utility with a small amount of resources.

(ii) Egalitarian welfare: Here, we choose the allocation a which maximizes the egalitarian welfare (minimum of all utilities), i.e. $a = \text{argmax}\, W_\text{E}$, where

$$W_\text{E} = \min_{j\in\{1,\ldots,n\}} u_j(a_j, \ell_j) = \min_{j\in\{1,\ldots,n\}} u'_j(p_j(a_j, \ell_j)). \quad (2)$$

This allocates more resources to "struggling" jobs which need more resources to achieve large utility (Figure 5).

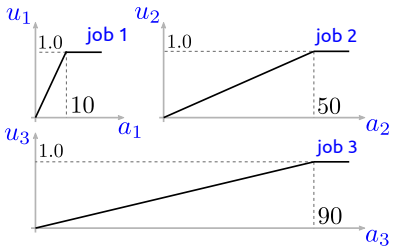
Demand-based policies: These policies apply when jobs have a well-defined SLO and it is possible to define its demand d_j. Such policies will compute allocations based on the demands of all jobs. This requires knowledge of the demand, which in turn depends on the performance mapping.

(iii) No justified complaints (NJC) fairness [16, 17, 28]: One class of demand-based policies which adopt the NJC fairness paradigm guarantee an equal share of R/n for each job. If the job's demand is larger than R/n, it is allocated at least (but possibly more than) this share. But, if the job's demand is smaller, the excess resources may be allocated to other jobs to improve overall resource usage. A user can have no justified complaints since they are either guaranteed to satisfy their SLOs or their utility will be larger than if they were to have R/n resources. To quantify this, we define the following metric. The term inside the minimum measures the utility achieved by job j with allocation a_j relative to the utility when using its fair share of R/n resources.

$$F_\text{NJC} = \min_{j\in\{1,\ldots,n\}} \frac{u_j(a_j, \ell_j)}{u_j(R/n, \ell_j)} = \min_j \frac{u'_j(p_j(a_j, \ell_j))}{u'_j(p_j(R/n, \ell_j))} \quad (3)$$

In contrast to metrics such as the Jain's index [31], F_NJC accounts for users' performance when evaluating fairness. This metric has a maximum value of 1. Below, we describe a demand-based policy [16] which achieves $F_\text{NJC} = 1$ while also using the resources efficiently as also shown in Figure 5.

An NJC policy: This policy proceeds iteratively. In the first round, it sets each user's "share" to be R/n. It allocates d_j to each user j for whom d_j is smaller than the share. If n'



Policy	allocation (utility) job 1		job 2		job 3		metrics W_S	W_E	F_{NJC}
Social welfare	10	(1.0)	50	(1.0)	0	(0.0)	0.67	0.0	0.0
Egalitarian welfare	4	(0.4)	20	(0.4)	36	(0.4)	0.4	0.4	0.4
NJC fair	10	(1.0)	25	(0.5)	25	(0.28)	0.59	0.28	1.0
Resource fair	20	(1.0)	20	(0.4)	20	(0.22)	0.54	0.22	1.0

Figure 5: Comparison of the three (oracular) fair allocation criteria described in §4.1.1 in a synthetic example with 60 CPUs. *Left:* Utility curves for three jobs. The y axis is the utility and the x-axis is the number of resources. For simplicity, we have ignored the loads and assumed that utilities increase linearly up to the demand. The total demand is 150, whereas only 60 resources are available. *Right:* The allocations and utilities for each job under the three criteria. We have also shown the W_S (1), W_E (2), and F_{NJC} (3) metrics for each policy.

users were allocated R' resources in the first round, in the second round it sets each user's share to be $(R-R')/(n-n')$. It repeats this until all the remaining users' demands are larger than their share. It then divides up the remaining resources equally among the remaining users. While this policy may not maximize any welfare, it achieves Pareto-efficient user utilities. Another advantage of this policy is that it is strategy-proof, i.e a user does not gain additional utility by falsely stating their demand [24, 28, 36].

This concludes our review of multi-tenant resource allocation objectives when performance mappings are known. We mention that prior work have used these objectives in various contexts with custom utilities. For instance, social welfare has been used in stream processing [35] and wireless networks [55], egalitarian welfare in video streaming [45], and several NJC policies are implemented in Mesos [29].

4.1.2 Online learning policies in Cilantro

We will now develop our online policies. Our policies will operate on lower and upper confidence bounds obtained from the load forecasters and performance learners instead of the direct estimates; doing so accounts for the uncertainty in the learned models and encourages a policy to conservatively explore the space of allocations until the estimates become accurate. Cilantro's policies will proceed sequentially in allocation rounds. On round r, Cilantro chooses an allocation $a^{(r)} = (a_1^{(r)}, \ldots, a_n^{(r)})$ based on the feedback from all jobs up to now and the specific scheduling objective.

Welfare-based online policies: For welfare-based policies, Cilantro adopts the optimism in the face of uncertainty (OFU) principle [7]. OFU stipulates that, to maximize an uncertain function, we should choose actions which maximize an upper confidence bound (UCB) on the function. Both theoretically and empirically, OFU is known to outperform other strategies which use direct estimates or those which are pessimistic (i.e. maximize lower confidence bound). An in-depth exploration of OFU is beyond the scope of this work, but we refer the reader to relevant literature (e.g. [6, 8, 25, 53]).

While OFU is a well established design paradigm, most OFU policies are designed for end-to-end systems which output a single reward signal. Adapting OFU for general welfare-based policies requires studying how the uncertainty in the performance and load translate to a UCB $\widehat{W}$ on the welfare W which we wish to maximize. Since W is non-decreasing in the utilities u_j, we can obtain a UCB for W by plugging in UCBs $\widehat{u}_j$ for the utility u_j, i.e $\widehat{W} = W(\widehat{u}_1, \ldots, \widehat{u}_n)$. Similarly, since u_j is non-decreasing in the performance we can obtain a UCB by plugging in a UCB $\widehat{p}_j$ for p_j, i.e $\widehat{u}_j = u'_j(\widehat{p}_j)$. This leads to the following choice of allocation on round r.

$$a^{(r)} = \operatorname*{argmax}_{a \in \mathcal{A}^{(r)}} W\left(u'_1\left(p_1\left(a_1, \widehat{\ell}_1\right)\right), \ldots, u'_1\left(p_1\left(a_n, \widehat{\ell}_n\right)\right)\right) \quad (4)$$

Above, since the exact load cannot be known, we conservatively over-estimate it via a UCB $\widehat{\ell}_j$ on the load. Here, $\mathcal{A}^{(r)}$ is the allocation space on round r which is defined by two constraints: first, the total allocation cannot be larger than R, i.e. $\sum_j a_j \leq R$; second, the current allocation cannot deviate too much from the previous allocation, i.e. $a_j^{(r-1)} - B \leq a_j \leq a_j^{(r-1)} + B$ for all j, where B is a parameter to be specified. We impose the second constraint since large changes to allocations can have unpredictable effects on a job's performance; moreover, they take a long time to actuate, resulting in unreliable feedback while resources are being scaled up/down.

To optimize (4), one can use any off-the-shelf optimizer such as evolutionary algorithms, hill climbing, or integer programming which can handle the linear constraints for $\mathcal{A}^{(r)}$. In our implementation, we used an evolutionary algorithm (details in the appendix). Finally, we describe instantiations of this principle for the two welfare-based policies we saw in §4.1.1.

(i) Cilantro-SW: To emulate the social welfare policy in §4.1.1, on round r, we use the UCB for $\hat{l}$ for load and $\hat{p}$ for performance. Thus, we choose an allocation

$$a^{(r)} = \operatorname*{argmax}_{(a_1,\ldots,a_n) \in \mathcal{A}^{(r)}} \sum_{j=1}^{n} u'_j\left(\widehat{p}_j(a_j, \widehat{\ell}_j)\right).$$

(ii) Cilantro-EW: To emulate the egalitarian welfare policy in §4.1.1, on round r, we choose an allocation

$$a^{(r)} = \operatorname*{argmax}_{(a_1,\ldots,a_n) \in \mathcal{A}^{(r)}} \min_{j \in \{1,\ldots,n\}} u'_j\left(\widehat{p}_j(a_j, \widehat{\ell}_i)\right),$$

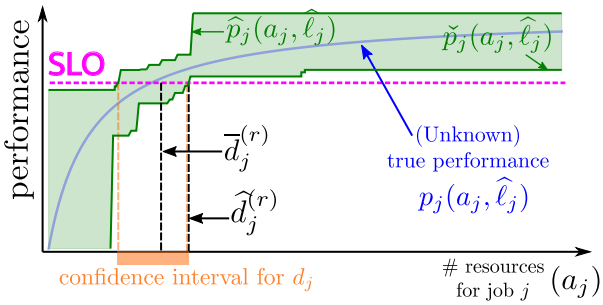


Figure 6: Illustration of Cilantro's uncertainty-aware demand-based policies. We first obtain a UCB $\widehat{\ell}_j$ from the load forecaster, which ensures that we have a conservative estimate on the job's load. In the figure, the x axis is the amount of resources a_j that could be allocated to job j. We show the SLO (pink), the slice of the unknown performance curve (blue) when the load is $\widehat{\ell}_j$, and the confidence region obtained from past data (green). The LCB $\check{p}_j$ and UCB $\widehat{p}_j$ on $p_j(a,\widehat{\ell}_j)$ are given by the lower and upper boundaries of the confidence region (solid green lines). A confidence interval for the demand (orange) can be obtained by the region where $\widehat{p}_j, \check{p}_j$ intersect the SLO line. To obtain a recommendation, we compute a UCB $\widehat{d}_j^{(r)}$ on the demand (where SLO intersects $\check{p}_j$) and $\overline{d}_j^{(r)}$ via equation (5).

Demand-based online policies: For demand-based policies, on round r, we will use the confidence intervals from the performance learners and load forecasters to obtain conservative recommendations $d_j^{(r)}$ for job j's demand. Then, we compute the allocations $a^{(r)}$ for this round by invoking the same demand-based policy with the recommended demands $\{d_1^{(r)},\ldots,d_n^{(r)}\}$ instead of the true demands.

Our method for obtaining demand recommendations is based on [36]. To describe this in more detail, observe that for demand-based policies it is sufficient to accurately estimate the demand well, i.e. it is not necessary to learn the entire performance mapping well. We have illustrated our strategy for obtaining the demand recommendation in Figure 6. First, we will denote by $\widehat{d}_i^{(r)}$, the UCB for the demand obtained as shown in Figure 6. As a conservative choice for this demand, we may wish to choose $\widehat{d}_i^{(r)}$ as the recommendation. However, we found that in practice this was overly conservative and the resulting allocations were very slow to adapt to feedback. Therefore, we also wish to use a more aggressive exploration strategy to reduce the uncertainty in our *demand*. We use:

$$\overline{d}_j^{(r)} = \operatorname*{argmax}_{a_j} \min\left(\widehat{p}_j(a_j,\widehat{\ell}_j) - \text{SLO},\ \text{SLO} - \check{p}_j(a_j,\widehat{\ell}_j)\right) \quad (5)$$

To illustrate this rule, consider Figure 6 where $\min(\widehat{p}_j - \text{SLO},\ \text{SLO} - \check{p}_j)$ is negative for large allocations when the performance LCB $\check{p}_j$ is larger than the SLO and for small allocations where the performance UCB $\widehat{p}_j$ is smaller than the SLO. By maximizing (5), we are choosing points inside the confidence interval for the demand where both $\check{p}_j, \widehat{p}_j$ are further away from the SLO; so if job j were to receive $\overline{d}_j^{(r)}$ resources, then we are most likely to reduce the demand uncertainty. However, choosing $\overline{d}_j^{(r)}$ as the recommendation can lead to overly aggressive exploration so our final recommendation $d_j^{(r)}$ is then obtained via,

$$d_j^{(r)} = \text{clip}\left(\beta\widehat{d}_j^{(r)} + (1-\beta)\overline{d}_j^{(r)},\ d_j^{(r-1)} - B,\ d_j^{(r-1)} + B\right) \quad (6)$$

Here, $\beta \in (0,1)$ is a parameter to trade-off between $\widehat{d}_i^{(r)}$ and $\overline{d}_i^{(r)}$. We clip this value between $d_j^{(r-1)} - B$ and $d_j^{(r-1)} + B$ to control wide deviations in resource allocations (similar to before). Next, we formally state Cilantro's instantiation of the demand-based NJC procedure described in §4.1.1.

(iii) Cilantro-NJC: Here, we simply compute the recommended demand via (5), and then invoke the NJC procedure described in §4.1.1, In §7.1 we show that Cilantro-NJC retains some of the strategy-proofness properties of NJC.

4.2 Microservice resource allocation

Now, we will look another use-case for Cilantro, where we wish to optimize an end-to-end performance metric p of an application composed of several interdependent microservices (jobs). Examples for p include the total throughput of the application, the negative P99 latency, or even any combination of the two. Here, the entire fixed set of resources is available to the application and must be allocated between microservices for to maximize p. There are two main differences in this setting when compared to the multi-tenant setting which introduces new challenges. First, while assuming jobs run by different users are independent is reasonable when we aim to optimize for fairness, this is no longer true now since microservices within an application may have complex dependency graphs (see Figure 12-Left). Second, while an application's performance is clearly tied to the performance of individual microservices, it is not possible to write it explicitly, as we did for the social or egalitarian welfare.

We overcome these challenges by modeling the end-to-end performance p as a direct function of the allocation to each microservice and the external load faced by the application. That is, we write $p(a,\ell)$, where, $a = (a_1,\ldots,a_n)$ is a vector of allocations for each microservice and ℓ is the external load on the application. On allocation round r, our online learning policy, which adopts the OFU principle, chooses an allocation vector which maximizes an upper confidence bound $\widehat{p}$ on the performance obtained from the performance learners:

$$a^{(r)} = \operatorname*{argmax}_{a \in \mathcal{A}^{(r)}} \widehat{p}(a,\ell). \quad (7)$$

While this circumvents accounting for individual microservice performance and dependency graphs, we now face the challenge of optimizing for an n–dimensional allocation with just one feedback signal. In contrast, in the multi-tenant setting we had more feedback (one for each job).

5 Discussion

We now present a discussion on Cilantro's operation under various adversarial conditions that may occur in deployment.

When online job feedback is unavailable. Cilantro provides three fallback options when online feedback is not available. First, Cilantro allows a user to use a profiled model (using historical data) instead of online feedback. Second, it allows using proxy metrics from the Kubernetes API instead of real-world performance. In such cases, a user should specify how these proxies are tied to their utility and/or demand. Third, if neither of these is possible, we allow the user to directly submit an estimate for their resource demand which will then be fed to the policy when determining allocations. In such cases, we assume that utility increases linearly up to the demand when computing allocations. We evaluate this fallback option in §7.3. Due to Cilantro's decoupled design, these fallback options can be effected with simple modifications to a job's performance learner.

Learning in unpredictable environments. Some situations, such as unexpected load spikes for web services or interference between jobs, are fundamentally hard to predict. Cilantro's uncertainty-aware design provides a degree of resilience against these unpredictable changes, as we show in it's robustness to noise in load and resource demand estimates in Section 7.3. However, continued extreme fluctuations in the loads can negatively impact Cilantro's performance. To avoid hysteresis when reallocating resources, future work can explore averaging loads over dynamically sized windows or including rules to temporarily override Cilantro's policy.

Limitations and Future Work. Cilantro currently supports allocating only a single resource type. In our current implementation, multiple resource types can be bundled into grouping units, such as VM SKUs with a fixed ratio of CPU, Memory and GPUs, which can then be scheduled by Cilantro. However, such bundling is not always possible, especially when different jobs have different resource requirements. Extending Cilantro to handle multiple resource types is possible for welfare-based policies. However, learning and optimization can be challenging since the search space is now very large. Another related limitation is that Cilantro cannot handle non-fungible resource types. Cilantro also does not support online learning versions of market-based resource allocation policies in the multi-tenant setting [39, 56, 63]. These are avenues for future work to improve Cilantro. Cilantro also assumes utilities increase with increasing resources, however some workloads may demonstrate inverse scaling, especially when allocated resources become fragmented across physical nodes. Future work can relax this assumption by applying learning techniques robust to non-convex utility shapes. We also note that Cilantro can support multiple SLO parameters (e.g., for an inference job, ensuring a minimum latency and accuracy) by wrapping them in a single utility function, and the design of such utility functions can be explored by future work.

6 Implementation

The Cilantro scheduler is implemented in 7600 lines of Python code, as a standalone scheduler for Kubernetes. Resource reallocation events are triggered by a timer-based event, which is raised every 2 minutes in our experiments. This window was chosen based on the fact that Kubernetes pods could be created and destroyed in 5-15 seconds. A 2 minute allocation round is long enough for the pod to reach its steady state that performance metrics from the job would be reliable, while at the same time frequent enough to adapt to changes in the load and learned performances.

To execute updated resource allocations received from policies, we horizontally scale the workloads by adding more replicas to their Kubernetes deployment. Newly created pods rely on the Kubernetes service discovery mechanism to connect to the workload's other servers. The workload is responsible for load balancing queries onto the new servers. Workloads write logs to a volume shared with the sidecar cilantro client. The client parses performance metrics and then publishes them to the scheduler over gRPC. These messages also act as heartbeats to inform liveness to the scheduler.

The frequency of performance feedback depends on the application and the environment. For instance, database serving jobs may report feedback multiple times in a minute, while ML training jobs may do so once every few minutes. To avoid bottlenecks, we use an asynchronous design for Cilantro where each component operates in a push or pull based framework. This allows high frequency components to operate at their maximum rate while allowing slower components, such as learners for low-frequency jobs or cluster managers, to be polled when required.

Specifying utilities and objectives. Utilities of jobs are calculated based on the performance metrics collected by the Cilantro clients in the last resource allocation round. To compute the utilities, application developers specify utility as a python method which operates on a list of floating point numbers representing the performance metrics observed in the previous resource allocation round. Similarly, the scheduling objective (e.g., social welfare from §4.1.1) is also defined by the cluster operator as a python method operating on the list of utilities from all jobs.

Learning models and load forecasters. For the multi-tenant setting, we used a tree-based binning estimator [8,27,36] with Lipschitz constant 10 for each job's resource-to-performance estimation. This is a simple and computationally efficient estimator, but does not work well in high dimensions. Therefore, for the microservices setting where we have a high dimensional estimation challenge, we use kernel ridge regression [59,65] with a Matern kernel with smoothness parameter

set to 2.5. In both settings, for the load forecasters, we use an autoregressive moving average (ARMA) model [41] with autoregressive order 1 and moving average order 1. Finally, all confidence bounds were computed at the 90% level, meaning that the probability that the true parameter lies between the upper and lower confidence bounds is 90%. We used the above learning models since they are simple and have few tunable hyperparameters. With Cilantro's modular design, these can be easily swapped with any other model as long as they provide reliable uncertainty estimates.

Other policy parameters: For all our policies, we set the parameter B which controls the deviation from the previous allocation to 10. For demand-based policies, we set the parameter β which trades off between conservative and aggressive exploration to $3/4$. For the welfare-based policies in §4.1 and the microservices use case in §4.2, we use evolutionary algorithms to optimize the UCBs. The exact implementation is described in the appendix.

7 Evaluation

We evaluate Cilantro in two settings described in §4.

1. In the multi-tenant setting, Cilantro's online learning policies, which do not start with any prior data, are competitive with oracular policies which have access to jobs' resource to performance mappings obtained after several hours of profiling. Moreover, they outperform 9 other baselines on the metrics outlined in §4.1.
2. In the microservices setting, Cilantro is able to support the completely different objective of minimizing end-to-end latency. It outperforms three other baselines and reduces the P99 latency to $\times 0.57$ that achieved by the next best performance-aware baseline.
3. In our microbenchmarks, we show that Cilantro's allocation policies are inexpensive, evaluate its fallback options when performance metrics are unavailable, and demonstrate its robustness to errors in feedback and choices for performance learner and forecaster models.

7.1 Multi-tenant cluster sharing

We first evaluate Cilantro's multi-tenant policies (§ 4.1.2) on a 1000 CPU cluster shared by 20 users.

Workloads. We use three classes of workloads—database querying, prediction serving and machine learning training—which are used to create multiple jobs. The database querying workload runs TPC-DS [43] queries on replicated instances of sqlite3 database and uses the query latency as the performance metric. The prediction serving workload runs queries on a ML model (random forest regressor) trained on the news popularity dataset [20]. The ML training workload trains a neural network on the naval propulsion [12] dataset using stochastic gradient descent. The database querying and prediction serving workloads use the query latency as the performance metric while ML training uses batch throughput to measure performance. Resource-performance mappings for informing the oracle baselines in §7.1 were obtained through offline profiling of all workloads. These profiles are visualized in Figure 7. More details of the workloads, including workload-specific parameters are available in the appendix.

Traces. Queries to the database and prediction serving workloads are dispatched by a trace-driven workload generator. We use the Twitter API [5] to collect a trace of tweet arrival rates at Twitter's Asia datacenters; to bring to parity with our cluster, we subsample the arrival rate by a factor of 10. For the ML training workload, we draw queries from an essentially infinite pool to create a constant stream of work.

Experimental set up. We use a cluster of 250 AWS m5.xlarge instances (4 vCPUs each). The Cilantro scheduler runs on its own dedicated m5.xlarge instance. We use the above 4 workloads to create 20 jobs as follows: 10 database jobs with P90, P90, P90, P90, P95, P95, P95, P95, P99, P99 latency SLOs of 2s; 3 prediction serving jobs with P90, P90, and P95 latency SLOs of 2s; 7 ML training jobs with throughput SLOs of 400, 400, 450, 450, 500, 500, and 500 QPS. To reflect settings where small SLO violations may be either critical or inconsequential, we discount the utility via one of the three options in Fig. 4 for each job. Detailed information on the users' jobs is given in the appendix. The estimated total amount of resources based on the median demand was 1637 CPUs; hence, even at full capacity, not all users can satisfy their SLOs. We evaluate all baselines for 6 hours.

7.1.1 Baselines

Oracular policies. We implement the three policies in §4.1.1 with oracular access to the true performance mappings (obtained by exhaustively profiling workloads for at least 4 hours). They are Oracle-SW, Oracle-EW, for maximizing social/egalitarian welfare and the Oracle-NJC fairness policy.

Cilantro policies. We evaluate Cilantro-SW, Cilantro-EW, and Cilantro-NJC, as described in Sec. 4.1.2.

Other heuristics. We implement four methods for fairness and maximizing welfare. While not based directly off specific prior work, such methods are common in the scheduling literature [13, 26]. Resource-Fair simply allocates an equal amount of resources to each job. EvoAlg-SW and EvoAlg-EW are evolutionary algorithms for social and egalitarian welfare; the same procedure used for Cilantro's welfare policies, but now operating directly on the performance metrics. Greedy-EW starts by allocating resources equally; on each round, it evaluates job utilities in the previous round and takes away one CPU each from the top half of the users who had high utility and allocates it to the bottom half.

Baselines from prior work. We adapt five feedback-driven methods from prior work - Ernest [58], Quasar [15], Minerva [45], Parties [10] and MIAD (Multiplicative-

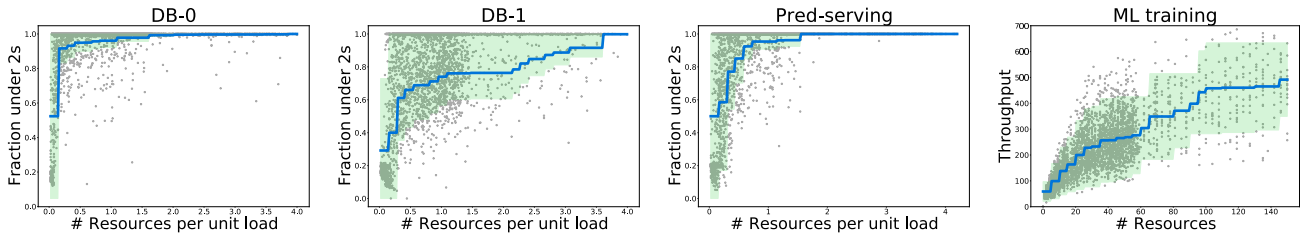


Figure 7: Performance vs resource-allocation-per-unit-load obtained after profiling the database querying, predicition serving and ML training workloads. The blue curve is the average performance value and the shaded region is the 2σ confidence interval. For the latency-based workloads (DB-0, DB-1, and prediction serving), we show the number of resources per unit load (arrival QPS) on the x-axis and the fraction of queries completed under 2s on the y-axis. For the ML training workload, we show the number of resources on the x axis the amount of data processed per second on the y-axis. To obtain accurate estimates, we sampled low resources allocations more densely.

Increase/Additive-Decrease) [11]. In particular, we note that applying the Parties notion of migration in our setting would imply moving the job to a different cluster or increasing the size of the cluster, both of which are beyond scope for this fixed cluster setting. Details on the specific adaptations are available in the appendix.

7.1.2 Results & Discussion

Evaluation on performance-aware fairness metrics. We first compare all 15 baselines on the social welfare (1), egalitarian welfare (2), and the NJC fairness criteria (3). Fig. 8 illustrates the results by plotting the time-averaged NJC fairness vs the two welfare criteria. Table 2 (in the appendix) tabulates these values explicitly with error bars. While the oracular methods perform best on their respective metrics, we find that the online learning policies in Cilantro come close to matching them. Resource-Fair achieves a perfect NJC score by definition, but performs poorly on social and egalitarian welfare as it is performance oblivious.

We found that Greedy-EW, Parties, and MIAD were sensitive to the amount by which we changed the allocations based on feedback; when tuning them, we found that they were either too slow or too aggressive when responding to load shifts. Next, the learning models used by Quasar and Ernest were not able to accurately estimate the demands in our experiment. Finally, the evolutionary baselines were inefficient, taking a long time to discover the optimal solution. They, however, were effective within Cilantro's welfare policies when you need to optimize a cheap analytically computable function as they can be run for several iterations.

Despite our general approach, Cilantro's policies are able to outperform Minerva and Greedy-EW which are designed specifically to maximize egalitarian welfare. It also outperforms generically designed evolutionary algorithms for the social and egalitarian welfare. While it may indeed be possible to design more efficient fine-tuned policies for a given objective, the flexibility provided by Cilantro's approach is beneficial to end users. It should not be surprising that Cilantro outperforms other systems such as Ernest, Quasar, Parties, and MIAD as our policies are designed to explicitly optimize for these objectives. *But this is precisely the goal of Cilantro*. End-users can declare their desired objective, and Cilantro will automatically derive policies to achieve them.

To illustrate how Cilantro improves with feedback, in Fig. 9, we have shown how the three objectives evolve over time for Cilantro's policies. Resource-Fair trivially achieves $F_{\text{NJC}} = 1$ at start since our initial allocation is always 50 CPUs to each job (i.e Resource-Fair). However, it does poorly on welfare due to poor cluster usage. The goal behind Cilantro-NJC is to achieve $F_{\text{NJC}} = 1$ while also achieving good cluster usage. This causes the initial drop in performance for Cilantro-NJC as it explores better allocations that still maximize F_{NJC}.

Table 2 presents the detailed results of our multi-tenant cluster resource sharing evaluation. This table adds a metric which measures the useful resource usage.

$$\text{Useful resource usage} = \sum_{j=1}^{m} \min(a_j, d_j) \tag{8}$$

Here, the d_j is user j's resource demand. This demand-based metric, measures how much *useful* work is being done by the cluster as allocations beyond the demand do not increase a user's utility (see Fig. 4). We find that Cilantro's policies achieve the maximum useful resource usage in their respective classes. This is because learning resource demands allows Cilantro to reallocate resources from jobs which have already achieved maximum utility to jobs which can benefit from increased resources.

Individual user utilities. To delve deeper into the trade-offs of the three paradigms discussed in §4.1, we have shown the individual user utilities achieved by these three policies in Fig. 10. We see that both the social and egalitarian welfare policies result in some users being worse off than receiving their fair allocation of $1000/20 = 50$ CPUs. This results in an NJC fairness violation. In contrast, in Cilantro-NJC, users are at most marginally worse off than their fair share. However, a third of the users achieve a noticeably higher utility than their fair share utility, with more than $3\times$ for a few of them. We also see that Cilantro-EW has maximized egalitarian welfare by taking resources away from those who achieve high utility and giving it to those who do not, while Cilantro-SW has

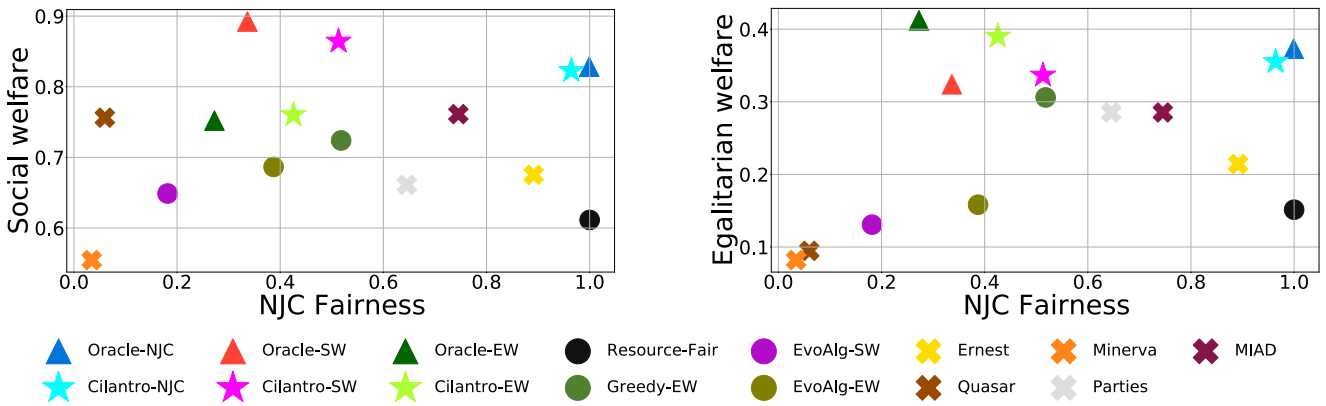


Figure 8: NJC fairness vs the social and egalitarian welfare (see §4.1.1) for all policies. We report the average value over the 6 hour period. Higher is better for all metrics, so closer to the top right corner is desirable. The Oracle-SW, Oracle-EW policies optimize for the social and egalitarian welfare when the performance mappings are known and Oracle-NJC achieves maximum fairness while improving cluster usage. The corresponding Cilantro policies are designed to do the same without a priori knowledge of the performance mappings.

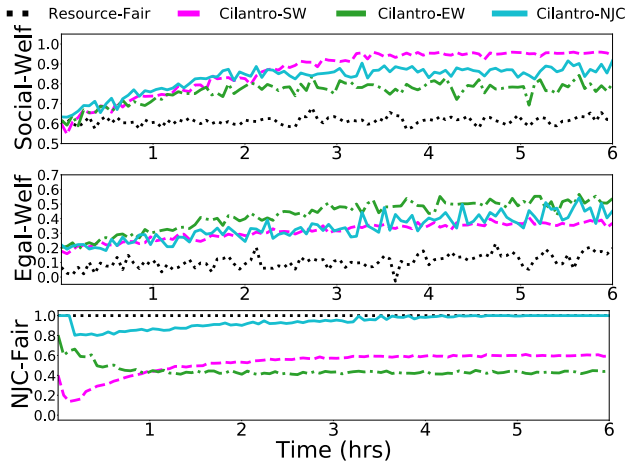


Figure 9: Convergence over time of social, egalitarian welfares and NJC fairness for the three Cilantro policies.

maximized social welfare by allocating more resources to jobs that can quickly achieve high utility.

Evaluating Strategy-proofness. We next evaluate Cilantro policies for strategy-proofness. A policy is said to be strategy-proof if an unscrupulous user cannot increase the utility of their job by misreporting their performance metrics to the scheduling policy. For this, we repeat the same experiment set up; all jobs behave exactly as before except the `db16` job which lies about its performance by either under-reporting by a factor $\times 1/2$, or over-reporting by a factor $\times 2$. By under-reporting, the user gives the impression that more resources are required to reach its SLO; in contrast, by over-reporting, a user is deceiving the scheduler to prioritize their job as they can achieve high utility with few resources. In Fig. 11, we report the utilities achieved by `db16` under these untruthful behaviors. We see that for Cilantro-NJC, the job's utility does not increase when over-reporting and decreases when under-reporting, leaving no incentive for the user to be untruthful. In contrast, for Cilantro-EW, a user stands to gain by under-reporting while for Cilantro-SW, they gain by over-reporting. While a theoretical study of such strategy-proofness properties is beyond the scope of this work, it is interesting to empirically observe that the strategy-proofness properties of NJC fairness policies are retained in Cilantro.

7.2 Resource allocation for Microservices

We now demonstrate the use of Cilantro to allocate resources for inter-dependent microservices serving an application. A query to the application triggers multiple queries to different microservices and the final result is returned to the user. Cilantro must observe a single end-to-end metric, the end-to-end query latency, and then allocate fixed cluster resources to different microservices to minimize the P99 latency of the application. We note that Cilantro does not require meta information about the microservices, such as their dependency and control flow graphs; Cilantro directly optimizes the end-to-end metric as described in §7.2.

Workload. We use the Hotel Reservation application from DeathStarBench [22]. It has 19 microservices, including 6 MongoDB databases, 3 memcached kv-stores and a nginx webserver running on a consul service mesh. The architecture is shown in Fig. 12-Left. Collectively, these microservices serve search, recommendation, rating, account management and geolocation queries from users. We use wrk2 [54] to process and submit the query workload provided in [22]. We measure the end-to-end latency of queries submitted to the frontend microservice. All microservices experiments are run on a 160 CPU cluster with 20 AWS m5.2xlarge instances.

Baselines. We compare Cilantro's end-to-end policy (§4.2) against three baselines. Resource-Fair always equally allocates the resources among microservices. EvoAlg is an evolutionary algorithm which optimizes for the P99 latency. ε-greedy randomly picks a new allocation with probability $1/3$,

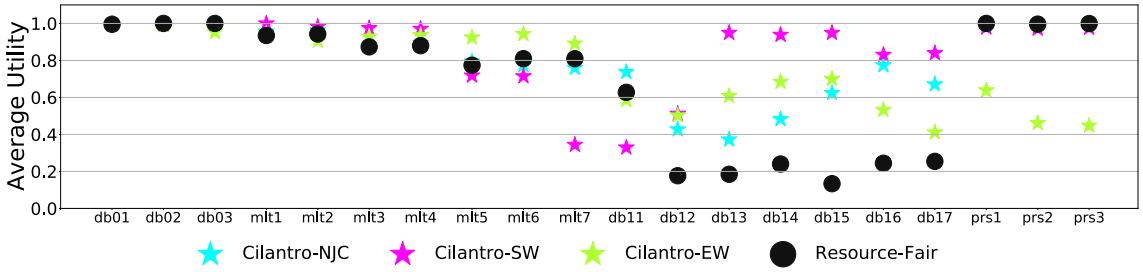


Figure 10: The average utility achieved by the 20 jobs for the three online learning methods in Cilantro and Resource-Fair. Here, `db0x`, `mltx`, `db1x`, and `prsx` refers to jobs using the DB-0, ML training, DB-1, and prediction serving workloads from § 7.1.

Policy	Social Welfare, (W_S)	Egalitarian Welfare (W_E)	NJC Fairness (F_{NJC})	Useful resource usage
Oracle-SW	0.892 ± 0.004	0.324 ± 0.008	0.336 ± 0.004	0.964 ± 0.002
Oracle-EW	0.752 ± 0.003	0.412 ± 0.007	0.272 ± 0.002	0.997 ± 0.000
Oracle-NJC	0.828 ± 0.002	0.373 ± 0.008	0.999 ± 0.000	0.991 ± 0.000
Cilantro-SW	$\mathbf{0.864 \pm 0.006}$	0.337 ± 0.013	0.513 ± 0.020	0.818 ± 0.012
Cilantro-EW	0.760 ± 0.007	$\mathbf{0.390 \pm 0.020}$	0.426 ± 0.037	$\mathbf{0.954 \pm 0.012}$
Cilantro-NJC	0.823 ± 0.002	0.355 ± 0.005	$\mathbf{0.964 \pm 0.006}$	0.931 ± 0.003
EvoAlg-SW	0.649 ± 0.017	0.131 ± 0.016	0.182 ± 0.048	0.671 ± 0.021
EvoAlg-EW	0.687 ± 0.011	0.158 ± 0.012	0.387 ± 0.040	0.700 ± 0.009
Resource-Fair	0.611 ± 0.002	0.151 ± 0.006	$\mathbf{1.000 \pm 0.000}$	0.766 ± 0.001
Greedy-EW	0.724 ± 0.005	0.306 ± 0.006	0.518 ± 0.009	0.882 ± 0.004
Ernest	0.675 ± 0.002	0.214 ± 0.005	0.891 ± 0.013	0.774 ± 0.002
Quasar	0.756 ± 0.002	0.095 ± 0.003	0.060 ± 0.003	0.706 ± 0.002
Minerva	0.555 ± 0.017	0.082 ± 0.006	0.034 ± 0.005	0.407 ± 0.023
Parties	0.661 ± 0.002	0.285 ± 0.006	0.645 ± 0.000	0.766 ± 0.001
MIAD	0.761 ± 0.002	0.285 ± 0.005	0.745 ± 0.000	0.766 ± 0.001

Table 2: The social welfare (1), egalitarian welfare (2), NJC fairness metric (3), and the effective resource usage (8) for all 13 methods. Higher is better for all four metrics, and the maximum and minimum possible values for all metrics are 1 and 0. The values shown in bold have achieve the highest value for the specific metric, besides the oracular policies. Resource-Fair has NJC fairness $F_{NJC} = 1$ by definition.

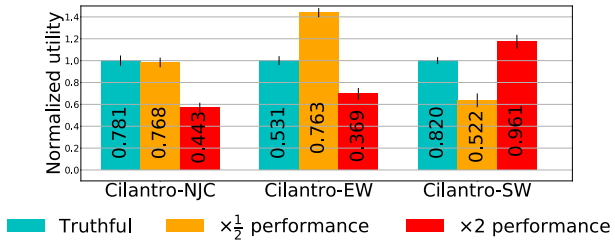


Figure 11: The utility of `db16` under the three online learning policies, when they report truthfully, when they under-report, and when they over-report. The plot normalizes with respect to truthful reporting, but the bars are annotated with the absolute value.

or uses the allocation with the smallest observed P99 latency with probability $2/3$.

Results and Discussion. Fig. 12 shows how the instantaneous and time-averaged P99 latency (computed in 30s intervals) evolves with time during the course of the experiment. Both Cilantro and EvoAlg explore early on (Fig. 12-Center), but as they find better values, exploration shrinks as they focus on testing more promising allocations. However, Cilantro's OFU-based online learning policy is able to do this more effectively than EvoAlg. ε-greedy explores aggressively even in later stages and is unable to adequately exploit good candidates it may have discovered in the early stages. Overall, Cilantro achieves a mean P99 of 525ms, compared to 930ms for EvoAlg, the next best baseline.

7.3 Microbenchmarks

Cilantro Overhead. Fig. 13-Left evaluates the time taken for Cilantro to process the feedback and compute the allocations for the three policies described in Sec. 4.1. This shows that Cilantro is fairly light-weight. For comparison, the average time it took to de-allocate a Kubernetes pod and assign it to a different job was on the order of 5-15s.

Unavailable performance metrics. In real-world situations, performance metrics of all users may not be available. We evaluate Cilantro's fallback defaults for such instances. We re-run the same experiment in §7.1, but for users `db01`, `mlt1`,

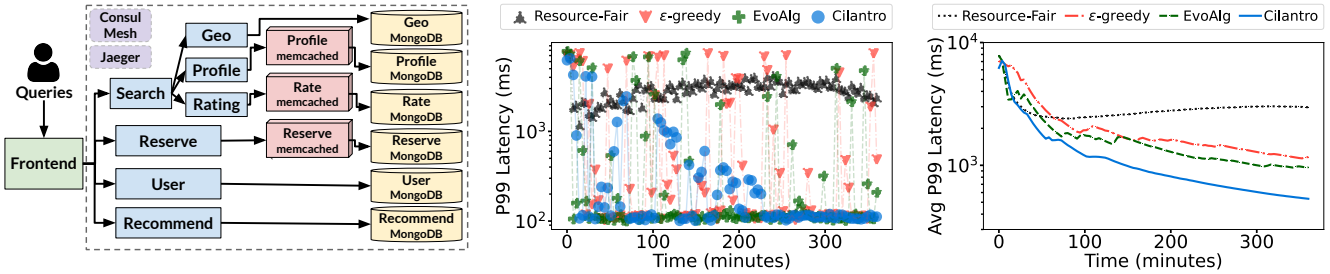


Figure 12: *Left:* Microservices architecture of the hotel reservation benchmark [22]. Blue boxes are business logic, red boxes are caching services, yellow boxes are databases and purple boxes are networking services. *Center:* Results for the microservices experiment comparing four methods on P99 latency over 6 hours, plotting the instantaneous P99 latency vs time. *Right:* The time-averaged P99 latency vs time.

Operation	Call time (s)
Model Update	0.0413 ± 0.0048
`get-alloc` call	
Cilantro-SW	2.8823 ± 0.3155
Cilantro-EW	2.1239 ± 0.0212
Cilantro-NJC	0.0081 ± 0.0016

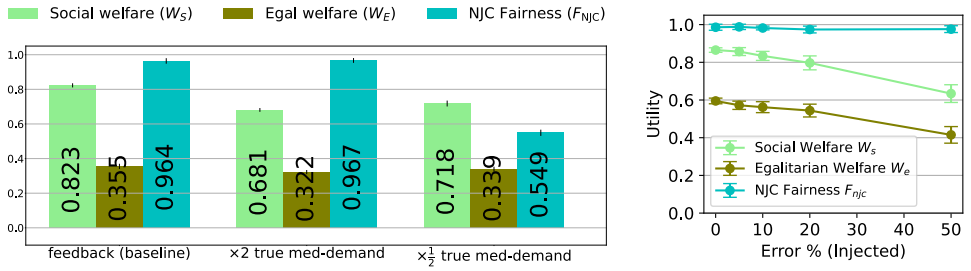


Figure 13: Cilantro microbenchmarks. *Left:* Mean time taken (in seconds) by Cilantro to update the performance model and for computing a new allocation for each of the three fixed cluster sharing policies. *Center:* Evaluation of Cilantro's fallback option, where users provide a demand value if they cannot report performance metrics. We evaluate Cilantro-NJC when 5 out of 20 users use this option. Since the true demand cannot be known, we use either half or twice the true demand under the median load from our profiled data. *Right:* The three performance metrics for Cilantro-NJC when we artificially introduce error to the confidence intervals of the performance and load.

`mlt2`, `db11`, and `prs1`, we manually set the demand as described in §5. Since the true demands are not known a priori, users might under- or overstate them. To reflect this, we first compute the true demand for each user under the median load from our profiled data. We evaluate Cilantro-NJC when these five users report either half this value as their demand or twice this value, when compared to providing feedback. Fig. 13-Center presents results on the three criteria given in §4.1. While the fallback options are worse than when reporting feedback, the failures are graceful. Cilantro is still able to learn from the remaining 15 users and achieve efficient allocations with only relatively small drops in social and egalitarian welfare. The NJC fairness criterion is significantly small when under-reporting since these 5 users will have been allocated at most half of their true demand and F_{NJC} (3) depends on the single worst fairness violation.

Robustness to choice of learners and feedback errors. While Cilantro's decoupled design aids with generality, it may be susceptible to the idiosyncrasies of the specific models used for the performance learners and load forecasters. Moreover, in many real environments, the feedback can be very noisy. To show that Cilantro is robust to both these effects, we perform the following microbenchmark in a synthetic 5 user environment (described in the Appendix) with the Cilantro-NJC policy. As both feedback noise and model idiosyncrasies can be modeled with inaccurate confidence intervals, we introduce increasing levels of noise (5%, 10%, 20%, 50%) to the upper and lower confidence bounds returned by the learners and forecasters. The results, given in Fig. 13-Right, show that the social and egalitarian welfare decrease gracefully with noise. Moreover, due to Cilantro-NJC's conservative approach for demand recommendations, the NJC fairness metric remains relatively high despite the noise.

8 Conclusion

We described Cilantro, a performance-aware framework for the allocation of a finite amount of resources among competing jobs. Our motivations were: *(i)* resource allocation policies should be performance-aware and based on real-time feedback in production environments, *(ii)* schedulers should accommodate diverse allocation objectives. We designed Cilantro to address these challenges by decoupling the performance learning from the policies and informing the policies of uncertainties in performance estimates, thus enabling the realization of several performance-aware policies in multi-tenant and microservices settings.

9 Acknowledgements

We thank the OSDI reviewers and our shepherd, Tim Harris, for their invaluable feedback. This work is in part supported by NSF CISE Expeditions Award CCF-1730628 and gifts from Astronomer, Google, IBM, Intel, Lacework, Microsoft, Nexla, Samsung SDS, Uber, and VMware.

References

[1] Amazon Compute Service Level Agreement. https://aws.amazon.com/compute/sla/, 2022.

[2] Hadoop Fair Scheduler. https://hadoop.apache.org/, 2022.

[3] Kubernetes api health endpoints | kubernetes. https://kubernetes.io/docs/reference/using-api/health-checks/, 2022.

[4] Ray dashboard — ray v1.7.0. https://docs.ray.io/en/latest/ray-dashboard.html, 2022.

[5] Twitter Streaming API. https://developer.twitter.com, 2022.

[6] Peter Auer. Using confidence bounds for exploitation-exploration trade-offs. *Journal of Machine Learning Research*, 3(Nov):397–422, 2002.

[7] Sébastien Bubeck and Nicolo Cesa-Bianchi. Regret analysis of stochastic and nonstochastic multi-armed bandit problems. *arXiv preprint arXiv:1204.5721*, 2012.

[8] Sébastien Bubeck, Rémi Munos, Gilles Stoltz, and Csaba Szepesvari. X-armed Bandits. *arXiv preprint arXiv:1001.4475*, 2010.

[9] Brendan Burns, Brian Grant, David Oppenheimer, Eric Brewer, and John Wilkes. Borg, omega, and kubernetes. *ACM Queue*, 14:70–93, 2016.

[10] Shuang Chen, Christina Delimitrou, and José F Martínez. Parties: Qos-aware resource partitioning for multiple interactive services. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 107–120, 2019.

[11] Dah-Ming Chiu and Raj Jain. Analysis of the increase and decrease algorithms for congestion avoidance in computer networks. *Computer Networks and ISDN systems*, 17(1):1–14, 1989.

[12] Andrea Coraddu, Luca Oneto, Aessandro Ghio, Stefano Savio, Davide Anguita, and Massimo Figari. Machine learning approaches for improving condition-based maintenance of naval propulsion plants. *Proceedings of the Institution of Mechanical Engineers, Part M: Journal of Engineering for the Maritime Environment*, 230(1):136–153, 2016.

[13] Daniel Crankshaw, Xin Wang, Guilio Zhou, Michael J Franklin, Joseph E Gonzalez, and Ion Stoica. Clipper: A low-latency online prediction serving system. In *14th USENIX Symposium on Networked Systems Design and Implementation (NSDI 17)*, pages 613–627, 2017.

[14] Christina Delimitrou and Christos Kozyrakis. Paragon: Qos-aware scheduling for heterogeneous datacenters. *ACM SIGPLAN Notices*, 48(4):77–88, 2013.

[15] Christina Delimitrou and Christos Kozyrakis. Quasar: resource-efficient and qos-aware cluster management. *ACM SIGPLAN Notices*, 49(4):127–144, 2014.

[16] Alan Demers, Srinivasan Keshav, and Scott Shenker. Analysis and simulation of a fair queueing algorithm. *ACM SIGCOMM Computer Communication Review*, 19(4):1–12, 1989.

[17] Danny Dolev, Dror G Feitelson, Joseph Y Halpern, Raz Kupferman, and Nathan Linial. No justified complaints: On fair sharing of multiple resources. In *proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, pages 68–75, 2012.

[18] Andrew D. Ferguson, Peter Bodik, Srikanth Kandula, Eric Boutin, and Rodrigo Fonseca. Jockey: Guaranteed job latency in data parallel clusters. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, page 99–112, New York, NY, USA, 2012. Association for Computing Machinery.

[19] Andrew D. Ferguson, Peter Bodik, Srikanth Kandula, Eric Boutin, and Rodrigo Fonseca. Jockey: Guaranteed job latency in data parallel clusters. In *Proceedings of the 7th ACM European Conference on Computer Systems*, EuroSys '12, page 99–112, New York, NY, USA, 2012. Association for Computing Machinery.

[20] Kelwin Fernandes, Pedro Vinagre, and Paulo Cortez. A proactive intelligent decision support system for predicting the popularity of online news. In *Portuguese Conference on Artificial Intelligence*, 2015.

[21] Brad Fitzpatrick. Distributed caching with memcached. *Linux journal*, 124, 2004.

[22] Yu Gan, Yanqi Zhang, Dailun Cheng, Ankitha Shetty, Priyal Rathi, Nayan Katarki, Ariana Bruno, Justin Hu, Brian Ritchken, Brendon Jackson, Kelvin Hu, Meghna Pancholi, Yuan He, Brett Clancy, Chris Colen, Fukang Wen, Catherine Leung, Siyuan Wang, Leon Zaruvinsky, Mateo Espinosa, Rick Lin, Zhongling Liu, Jake Padilla, and Christina Delimitrou. An open-source benchmark suite for microservices and their hardware-software implications for cloud and edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '19, page 3–18, New York, NY, USA, 2019. Association for Computing Machinery.

[23] Ali Ghodsi, Vyas Sekar, Matei Zaharia, and Ion Stoica. Multi-resource Fair Queueing for Packet Processing. In *Proceedings of the ACM SIGCOMM 2012 conference on*

Applications, technologies, architectures, and protocols for computer communication, pages 1–12, 2012.

[24] Ali Ghodsi, Matei Zaharia, Benjamin Hindman, Andy Konwinski, Scott Shenker, and Ion Stoica. Dominant resource fairness: Fair allocation of multiple resource types. In *Nsdi*, volume 11, pages 24–24, 2011.

[25] Alkis Gotovos. Active learning for level set estimation. Master's thesis, Eidgenössische Technische Hochschule Zürich, Department of Computer Science,, 2013.

[26] Robert Grandl, Mosharaf Chowdhury, Aditya Akella, and Ganesh Ananthanarayanan. Altruistic scheduling in multi-resource clusters. In *12th USENIX symposium on operating systems design and implementation (OSDI 16)*, pages 65–80, 2016.

[27] Jean-Bastien Grill, Michal Valko, and Rémi Munos. Black-box optimization of noisy functions with unknown smoothness. In *Advances in Neural Information Processing Systems*, pages 667–675, 2015.

[28] Avital Gutman and Noam Nisan. Fair allocation without trade. *arXiv preprint arXiv:1204.4286*, 2012.

[29] Benjamin Hindman, Andy Konwinski, Matei Zaharia, Ali Ghodsi, Anthony D Joseph, Randy H Katz, Scott Shenker, and Ion Stoica. Mesos: A platform for fine-grained resource sharing in the data center. In *NSDI*, volume 11, pages 22–22, 2011.

[30] Michael Isard, Vijayan Prabhakaran, Jon Currey, Udi Wieder, Kunal Talwar, and Andrew Goldberg. Quincy: fair scheduling for distributed computing clusters. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 261–276, 2009.

[31] Raj Jain, Dah-Ming Chiu, and W. Hawe. A quantitative measure of fairness and discrimination for resource allocation in shared computer systems. *CoRR*, cs.NI/9809099, 1998.

[32] Junchen Jiang, Vyas Sekar, and Hui Zhang. Improving fairness, efficiency, and stability in http-based adaptive video streaming with festive. In *Proceedings of the 8th international conference on Emerging networking experiments and technologies*, pages 97–108, 2012.

[33] Sangeetha Abdu Jyothi, Carlo Curino, Ishai Menache, Shravan Matthur Narayanamurthy, Alexey Tumanov, Jonathan Yaniv, Ruslan Mavlyutov, Inigo Goiri, Subru Krishnan, Janardhan Kulkarni, and Sriram Rao. Morpheus: Towards automated slos for enterprise clusters. In *12th USENIX Symposium on Operating Systems Design and Implementation (OSDI 16)*, pages 117–134, Savannah, GA, November 2016. USENIX Association.

[34] Vasiliki Kalavri, John Liagouris, Moritz Hoffmann, Desislava Dimitrova, Matthew Forshaw, and Timothy Roscoe. Three steps is all you need: fast, accurate, automatic scaling decisions for distributed streaming dataflows. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 783–798, 2018.

[35] Faria Kalim, Le Xu, Sharanya Bathey, Richa Meherwal, and Indranil Gupta. Henge: Intent-driven multi-tenant stream processing. In *Proceedings of the ACM Symposium on Cloud Computing*, SoCC '18, page 249–262, New York, NY, USA, 2018. Association for Computing Machinery.

[36] Kirthevasan Kandasamy, Gur-Eyal Sela, Joseph E Gonzalez, Michael I Jordan, and Ion Stoica. Online learning demands in max-min fairness. *arXiv preprint arXiv:2012.08648*, 2020.

[37] Mamoru Kaneko and Kenjiro Nakamura. The nash social welfare function. *Econometrica: Journal of the Econometric Society*, pages 423–435, 1979.

[38] Frank P Kelly, Aman K Maulloo, and David KH Tan. Rate control for communication networks: shadow prices, proportional fairness and stability. *Journal of the Operational Research society*, 49(3):237–252, 1998.

[39] Kevin Lai, Lars Rasmusson, Eytan Adar, Li Zhang, and Bernardo A Huberman. Tycoon: An implementation of a distributed, market-based resource allocation system. *Multiagent and Grid Systems*, 1(3):169–182, 2005.

[40] Avinash Lakshman and Prashant Malik. Cassandra: a decentralized structured storage system. *ACM SIGOPS Operating Systems Review*, 44(2):35–40, 2010.

[41] Spyros Makridakis and Michele Hibon. Arma models and the box–jenkins methodology. *Journal of forecasting*, 16(3):147–163, 1997.

[42] Jeonghoon Mo and Jean Walrand. Fair end-to-end window-based congestion control. *IEEE/ACM Transactions on networking*, 8(5):556–567, 2000.

[43] Raghunath Othayoth Nambiar and Meikel Poess. The making of tpc-ds. In *Proceedings of the 32nd International Conference on Very Large Data Bases*, VLDB '06, page 1049–1058. VLDB Endowment, 2006.

[44] Deepak Narayanan, Keshav Santhanam, Fiodar Kazhamiaka, Amar Phanishayee, and Matei Zaharia. Heterogeneity-Aware cluster scheduling policies for deep learning workloads. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 481–498. USENIX Association, November 2020.

[45] Vikram Nathan, Vibhaalakshmi Sivaraman, Ravichandra Addanki, Mehrdad Khani, Prateesh Goyal, and Mohammad Alizadeh. End-to-end transport for video qoe fairness. In *Proceedings of the ACM Special Interest Group on Data Communication*, pages 408–423. 2019.

[46] Hiep Chi Nguyen, Zhiming Shen, Xiaohui Gu, Sethuraman Subbiah, and John Wilkes. Agile: Elastic distributed resource scaling for infrastructure-as-a-service. In *International Conference on Automation and Computing*, 2013.

[47] Kay Ousterhout, Patrick Wendell, Matei Zaharia, and Ion Stoica. Sparrow: Distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, page 69–84, New York, NY, USA, 2013. Association for Computing Machinery.

[48] R. H. Patterson, G. A. Gibson, E. Ginting, D. Stodolsky, and J. Zelenka. Informed prefetching and caching. *SIGOPS Oper. Syst. Rev.*, 29(5):79–95, dec 1995.

[49] Haoran Qiu, Subho S. Banerjee, Saurabh Jha, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. FIRM: An intelligent fine-grained resource management framework for SLO-Oriented microservices. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 805–825. USENIX Association, November 2020.

[50] Charles Reiss, Alexey Tumanov, Gregory R. Ganger, Randy H. Katz, and Michael A. Kozuch. Heterogeneity and dynamicity of clouds at scale: Google trace analysis. In *Proceedings of the Third ACM Symposium on Cloud Computing*, SoCC '12, New York, NY, USA, 2012. Association for Computing Machinery.

[51] Krzysztof Rzadca, Pawel Findeisen, Jacek Swiderski, Przemyslaw Zych, Przemyslaw Broniek, Jarek Kusmierek, Pawel Nowak, Beata Strack, Piotr Witusowski, Steven Hand, et al. Autopilot: workload autoscaling at google. In *Proceedings of the Fifteenth European Conference on Computer Systems*, pages 1–16, 2020.

[52] Konstantin Shvachko, Hairong Kuang, Sanjay Radia, and Robert Chansler. The hadoop distributed file system. In *2010 IEEE 26th symposium on mass storage systems and technologies (MSST)*, pages 1–10. Ieee, 2010.

[53] Jasper Snoek, Hugo Larochelle, and Ryan P Adams. Practical bayesian optimization of machine learning algorithms. *Advances in neural information processing systems*, 25, 2012.

[54] Gil Tene. giltene/wrk2: A constant throughput, correct latency recording variant of wrk. https://github.com/giltene/wrk2. (Accessed on 04/19/2022).

[55] Yan Kyaw Tun, Nguyen H Tran, Duy Trong Ngo, Shashi Raj Pandey, Zhu Han, and Choong Seon Hong. Wireless network slicing: Generalized kelly mechanism-based resource allocation. *IEEE Journal on Selected Areas in Communications*, 37(8):1794–1807, 2019.

[56] Hal R Varian. Equity, envy, and efficiency. 1973.

[57] Vinod Kumar Vavilapalli, Arun C Murthy, Chris Douglas, Sharad Agarwal, Mahadev Konar, Robert Evans, Thomas Graves, Jason Lowe, Hitesh Shah, Siddharth Seth, et al. Apache hadoop yarn: Yet another resource negotiator. In *Proceedings of the 4th annual Symposium on Cloud Computing*, page 5. ACM, 2013.

[58] Shivaram Venkataraman, Zongheng Yang, Michael Franklin, Benjamin Recht, and Ion Stoica. Ernest: Efficient performance prediction for large-scale advanced analytics. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 363–378, 2016.

[59] Max Welling. Kernel ridge regression. *Max Welling's classnotes in machine learning*, pages 1–3, 2013.

[60] John Wilkes. *Utility Functions, Prices, and Negotiation*, chapter 4, pages 67–88. John Wiley and Sons, Ltd, 2009.

[61] Wencong Xiao, Romil Bhardwaj, Ramachandran Ramjee, Muthian Sivathanu, Nipun Kwatra, Zhenhua Han, Pratyush Patel, Xuan Peng, Hanyu Zhao, Quanlu Zhang, et al. Gandiva: Introspective cluster scheduling for deep learning. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 595–610, 2018.

[62] Matei Zaharia, Mosharaf Chowdhury, Michael J Franklin, Scott Shenker, Ion Stoica, et al. Spark: Cluster computing with working sets. *HotCloud*, 10(10-10):95, 2010.

[63] Seyed Majid Zahedi, Qiuyun Llull, and Benjamin C Lee. Amdahl's law in the datacenter era: A market for fair processor allocation. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 1–14. IEEE, 2018.

[64] Yanqi Zhang, Weizhe Hua, Zhuangzhuang Zhou, G Edward Suh, and Christina Delimitrou. Sinan: Ml-based and qos-aware resource management for cloud microservices. In *Proceedings of the 26th ACM International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 167–181, 2021.

[65] Yuchen Zhang, John Duchi, and Martin Wainwright. Divide and conquer kernel ridge regression. In *Conference on learning theory*, pages 592–617. PMLR, 2013.

[66] Hang Zhu, Kostis Kaffes, Zixu Chen, Zhenming Liu, Christos Kozyrakis, Ion Stoica, and Xin Jin. Racksched: A microsecond-scale scheduler for rack-scale computers. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, pages 1225–1240. USENIX Association, November 2020.

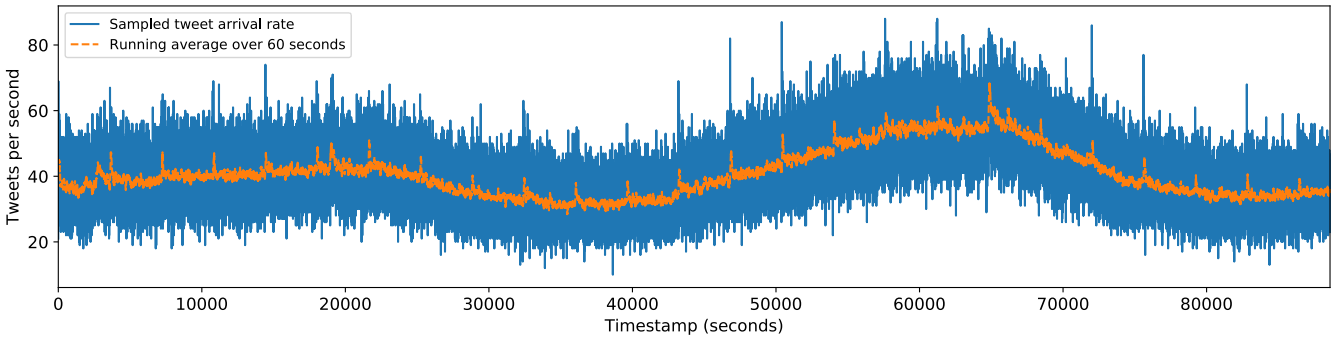


Figure 14: Sampled query arrival rate from the twitter trace collected over the duration of a day.

A Experiment Addendum

A.1 Workload description

Database querying: We use the TPC-DS [43] benchmark suite as the workload backed by replicated instances of sqlite3 database. From the TPC-DS query set, we created two workloads (setting scale factor to 100): DB-0, which had queries that completed in under 100 ms and DB-1 which had queries that had a completion time between 100 and 300 ms. When a query is requested, we randomly pick a relevant query and dispatch it according to the trace. The performance metric of interest is query latency.

Prediction serving: In prediction serving [13], a job processes arriving queries to output a prediction, usually obtained via a machine learning model. In our set up, we use a random forest regressor as the model and the the news popularity dataset [20] for training and test queries in a 50:50 split. Queries are picked randomly from the test set and issued in batches of 4. The metric of interest is the serving latency.

ML training: We use CPUs to train a neural network with four hidden layers of size 64 each. We train our model on the naval propulsion [12] dataset using stochastic gradient descent (SGD). Each task in this workload consists of training a batch of 16 points for 100 iterations. The performance metric of interest here is the batch throughput.

A.2 Environment details

Workload traces. As described in §7.1, we use traces collected from twitter to generate traffic patterns for our workloads. The query arrival rate of this trace is visualized in Figure 14.

Multi-tenant cluster jobs setup. For the multi-tenant cluster resource sharing evaluation, we setup 20 jobs with different workloads and SLOs as described in in §7.1. Table 3 details the exact SLO and utility function for each job. The utility function for each job is either of `linear`, which directly maps performance to utility (Figure 4(a)), `sqrt`, which performs a sublinear mapping of performance to utility (Figure 4(b)), or `quadratic`, which performs a superlinear mapping of performance to utility (Figure 4(c)).

TPC-DS Query Binning. The queries used for the db serving workload in §7.1 were selected from the TPC-DS benchmark suite. The TPC-DS suite consists of 99 query templates out of which 27 were not compatible with the sqlite dialect and were discarded. The remainder were binned according to their mean latency when measured on a AWS m5.2xlarge instance. The chosen query types and their ids are listed in Table 4

A.3 Baselines from prior work

Here we describe the specific implementation of prior work baselines used in Section 7.

1) Ernest [58]: Ernest uses a featurized linear model to estimate the time taken to run a job. We use this estimate to approximate the resource demand to meet the job's SLO. On each round, we use the estimated demand as inputs to NJC to compute the allocations.
2) Quasar [15]: Quasar uses collaborative filtering to estimate a job's resource demand, which we use as inputs to NJC to compute the allocations. We do not incorporate mechanisms for vertical scaling and workload co-location described in [15] to be consistent across all methods.

Job and SLO Type	Job Name	SLO	Utility Function
Database Serving (Latency)	db01	0.9	linear
	db02	0.9	linear
	db03	0.95	sqrt
	db11	0.9	linear
	db12	0.9	quadratic
	db13	0.95	quadratic
	db14	0.95	linear
	db15	0.95	quadratic
	db16	0.99	quadratic
	db17	0.99	sqrt
Prediction Serving (Latency)	prs1	0.9	linear
	prs2	0.9	sqrt
	prs3	0.95	sqrt
ML Training (Throughput)	mlt1	400	sqrt
	mlt2	400	sqrt
	mlt3	450	linear
	mlt4	450	linear
	mlt5	500	quadratic
	mlt6	500	quadratic
	mlt7	500	quadratic

Table 3: SLO and utility functions used for jobs in experiments in §7.1. For Latency based SLOs, the SLO implies the fraction of queries that completed under 2 seconds. For Throughput based SLOs, the SLO is the desired query rate, measured in queries per second.

Query Bin	TPC-DS Query Ids	Mean Execution Time (s)
db0	93, 91, 92, 45, 85, 15, 32	0.28
db1	90, 84, 8, 55, 96, 81, 79	0.67

Table 4: Details of the bins created from TPC-DS queries. Each user's workload is generated using these bins. Execution time is profiled on a SQLite3 database running on AWS m5.2xlarge instance with one allocated CPU core.

3) Minerva [45]: Minerva sets the allocation for job j at each step to be proportional to a_j/u_j where a_j and u_j are the allocation and utility at the previous round.

4) Parties [10]: Parties upsizes the allocation for a job if it violates or is close to violating the SLO, downsizes the allocation if the job comfortably satisfies the SLO, and otherwise does nothing. If the SLOs of all jobs cannot be met, it evicts the job from the server. As eviction is not an option in our setting we use the Parties logic to compute the demands which are then fed to NJC to obtain the allocations. For upsizing, we increase the demand by 20 CPUs and for downsizing, we decrease it by 5. These parameters were tuned so that the policy did reasonably well on all three metrics.

5) MIAD (Multiplicative-Increase/Additive-Decrease) [11]: This is inspired by TCP congestion control. If a user's job violates the SLO, we increase its demand by $1.5\times$ the current allocation, and if it satisfies the SLO, we set the demand to be one minus the current allocation. We then invoke NJC to compute the allocation for the next round. These parameters were tuned so that the policy did reasonably well on all three metrics.

A.4 Evolutionary Algorithm

We describe the evolutionary algorithm used in all of our experiments, i.e to optimize the profiled information for the oracular welfare polices, to optimize the upper confidence bounds for the learning policies in §4.1.2 and§4.2, and the evolutionary algorithm baselines in §7.1 and§7.2. The input to the algorithm is a data source which the algorithm can query using an allocation and obtain a feedback signal. This data source can either be a cheap analytically computable function available in memory, as is the case for the oracles and learning polices, or an expensive experiment, as is the case when used as a baseline to directly

optimize for performance. The algorithm maintains a hash table mapping allocations to mean observed signal values. When it receives feedback for an allocation, it updates the mean value if the allocation has already been tried, or it creates a new entry and stores the feedback.

Our evolutionary algorithm proceeds as follows. In has an initialization phase of 10 rounds. In the first 2 rounds, it always queries a resource-fair allocation. In the remaining 8 rounds, it queries a random allocation a such that $\sum_{j=1}^{n} a_j = R$. On each subsequent round, it chooses a random allocation in the above manner with probability 0.1. With probability 0.9, it samples one of the existing allocations in the hash table based on the mean feedback value, performs a mutation operation, and queries the new allocation obtained via the mutation. We now to describe these two steps.

- *Sampling:* Let $\{(a_i, y_i\}_i$ be the (allocation, mean feedback) pairs in the hash table. Let m, s denote the man and standard deviation of the $\{y_i\}$ values. We sample a_i with probability proportional to $\exp\big((y_i - m)/s\big)$.
- *Mutation:* The mutation operation is composed of a sequence of steps to modify a given allocation a. At each step, we randomly sample one job j which has an allocation of at least 2 CPUs; we then sample any other job $k \neq j$; we then decrease j's allocation by 1 and increase k's allocation by 1. The number of steps is chosen uniformly at random between 1 and 20.

A.5 Other experimental details

Synthetic environment for robustness microbenchmark: For the microbenchmark in Fig. 13(left), we use 5 users whose load is obtained by the same twitter trace from the experiments, and whose synthetic performance function is given by $p_j(a, \ell) = 1/(1 + e^{-(a/\ell - b_j)})$, where a is the allocation and ℓ is the load. For the 5 users, we set $b_j \in \{0.1, 0.3, 0.5, 0.7, 0.9\}$. We set the SLO to be 0.95 for all users (note that $0 \leq p_j \leq 1$. As the stochastic observation, we sample a Gaussian with standard deviation 0.2.