Proceedings of the Fifth Conference on
Human Computation and Crowdsourcing
(HCOMP 2017)

Cioppino: Multi-Tenant Crowd Management

Daniel Haas
AMPLab, UC Berkeley
dhaas@cs.berkeley.edu

Michael J. Franklin
The University of Chicago
mjfranklin@uchicago.edu

Abstract

Embedding human computation in systems for data analysis improves the quality of the analysis, but can significantly impact the end-to-end cost and performance of the system. Recent work in crowdsourcing systems attempts to optimize for performance, but focuses on single applications running homogeneous tasks. In this work, we introduce Cioppino, a system that accounts for human factors that can affect performance when running multiple applications in parallel. Cioppino uses a queueing model to represent the pool of workers, and leverages techniques for autoscaling used in cloud computing to adaptively adjust the pool size. Its model also accounts for worker abandonment, and automatically shifts workers between applications to improve performance and match workers with tasks they enjoy most. Our evaluation of Cioppino in simulation on traces extracted from a realtime crowd system running on Amazon's Mechanical Turk demonstrates a 19$\times$ reduction in cost, a 20% increase in throughput, and a 2$\times$ increase in worker preference for assigned tasks as compared to state-of-the-art crowd management strategies.

1 Introduction

When performance is a concern for crowd-powered systems, the slowness and unpredictability of recruiting workers can make sending tasks to platforms like MTurk impractical. Instead, workers can be recruited into *retainer pools* before work becomes available, and paid to wait if there are no available tasks (Bigham et al. 2010; Bernstein et al. 2011; Haas et al. 2015b). Most work in low-latency crowd systems has assumed that tasks are short and homogeneous. However, at organizations that rely heavily on human computation for data analysis, workers often work on a variety of tasks of varying difficulty and length (Marcus and Parameswaran 2015). These tasks range from simple binary question answering, such as labeling training data for machine learning models (Mozafari et al. 2014; Sheng, Provost, and Ipeirotis 2008; Cheng and Bernstein 2015) or identifying duplicate records for entity resolution (Gokhale et al. 2014; Wang et al. 2012), to complex work such as data extraction (Haas et al. 2015a), text translation (Callison-Burch 2009) or video analysis (Chen et al. 2009).

A system that uses the crowd to process heterogeneous tasks requested for different use cases has been called a *multi-tenant crowd system* (Difallah, Demartini, and Cudré-Mauroux 2016). Optimizing the performance of a such a system presents unique challenges, as illustrated by the following example.

Example 1 *Company X maintains a searchable database of products with public APIs, and has hired a crowd retainer pool of workers on a popular crowd platform to improve the quality of their data and services. Some workers monitor and filter incoming data containing explicit content, others extract structured records from newly scraped data sources, and still others provide search relevance feedback to improve API results. Though Company X currently hires and trains workers separately for each of these tasks, they often find that spikes and lulls in the volume of work leave workers overloaded or idle.*

In this paper, we address the challenges of Example 1 head on. How should Company X size their crowd to meet demand? Can the crowd be resized adaptively? Should workers be trained for just one type of work, or moved between task types to meet demand despite the training overhead? Following existing work on real-time crowdsourcing (Bernstein et al. 2012), we model our crowd pool as a queueing system where crowd workers process incoming tasks as they arrive. An important insight is that this queueing system bears many similarities to a cloud-based software-as-a-service (Saas) provider. SaaS systems handle requests to multiple service endpoints while attempting to maintain high overall throughput at low cost. To do so, they leverage techniques such as elastic cluster resizing (Ali-Eldin, Tordsson, and Elmroth 2012), handling of nodes that fail or leave a cluster (Dean and Ghemawat 2008), and resource sharing between applications that run on the same cluster (Ghodsi et al. 2011). A multi-tenant crowd system faces many analogous challenges: sizing the worker pool for the task workload, reacting when workers abandon the system, and ensuring that workers are assigned tasks that they enjoy. As a result, existing approaches to cloud service management provide insight into managing crowds.

However, because workers are human beings, we cannot treat them as idealized processors. Unlike CPUs, human crowd workers abandon tasks due to fatigue, confusion, or boredom. Individual workers exhibit varying preferences and skill for different tasks, and improve over time as they become familiar with their work. When modeling a crowdsourced computation system, these human factors simply cannot be ignored.

In this work, we explicitly model a subset of human factors which matter for system performance, and simulate multi-tenant crowd workloads to demonstrate the advantages of our model. The state-of-the-art in low-latency crowd systems does not automatically adapt the workforce size to workload changes, responds to worker abandonment reactively as workers leave, and either ignores under-utilized workers or assumes they have a static set of skills and cannot be trained on new tasks.

In contrast, we describe Cioppino, a system that explicitly models worker abandonment, task preferences, and training overheads, and addresses them with three novel techniques. *Pool elasticity* dynamically resizes the pool of workers to meet application demand without over-recruiting, leveraging techniques from the cloud autoscaling literature. *Pool stability* models the problem of worker abandonment. It maintains a worker pool of constant size by automatically recruiting new workers in a fashion adaptive to the rate of worker departure. *Pool balance* shares idle workers among applications to maximize throughput, balancing application demands against worker preferences.

Together, these techniques demonstrate that adjusting simple models to account for human behavior can improve the performance of multi-tenant crowd systems that support multiple concurrent crowd applications. We evaluate Cioppino in simulation using both synthetic data and a trace from a live crowd pool deployment and show a 19× decrease in cost over the crowd worker management used in today's systems, while doubling worker preference for the tasks they are assigned and improving throughput by 20%.

2 The CIOPPINO System

In this section, we survey related work in the context of the considerations that drive the design of crowd systems, then describe the design of Cioppino.

2.1 Tradeoffs in crowd system design

Designers of performant human computation systems must resolve a key tension between human behavior and programmatic abstraction. On the one hand, we would like to build programmable, low-latency, low-cost, high throughput systems that treat crowd workers as abstract computational resources. On the other, we must acknowledge that crowd workers are not CPUs, and that there are human factors that will confound performance if we don't take them into account. As a result, a good crowd worker abstraction must model enough of these factors to avoid surprises in performance, while not falling down the rabbit-hole of attempting to capture all of human behavior. Here, we list some important considerations that should be taken into account by crowd system designers, and survey related work that has addressed them. This paper focuses primarily on tradeoffs that improve performance by increasing throughput while accommodating worker preferences.

Application domain. The ease with which human workers understand and complete tasks of a specific type can affect system performance. For example, microtasks such as image labeling where the result is objectively correct or incorrect, the output data is well structured, and the data domain is easily understood by most workers can be incorporated into systems that use simple API calls to show workers tasks and aggregate their results. Systems that intentionally decompose complex problems into microtasks (Bernstein et al. 2010; Little 2009; Kittur et al. 2011) or those that recruit experts who better understand the problem domain (Retelny et al. 2014; Kulkarni et al. 2014) explicitly address this issue.

Desired latency and throughput. When workers are processing tasks at high speed, they will not behave as consistently as CPUs. It is well-known that rather than working away as fast as they can, crowd workers take breaks, exhibit fatigue or lack of attention to tasks, and eventually abandon the system entirely (Rzeszotarski et al. 2013; Bernstein et al. 2012). If latency is an important system metric and individual workers complete many tasks, these human factors must be taken into account. For example, inserting microbreaks (Rzeszotarski et al. 2013) can keep workers better focused during actual working time, and straggler mitigation or pool maintenance (Haas et al. 2015b; Ramesh et al. 2012) can limit the effect of workers who are responding unusually slowly. Additionally, careful task design can help reduce the latency of the crowd (Krishna et al. 2016; Marcus et al. 2012).

Desired output quality. Not all workers are equally good or fast at performing a given task, and workers exhibit familiarization effects over time (Ipeirotis 2010). A large body of work on quality control for crowdsourcing attempts to model and address this issue. For example, algorithms for combining multiple worker responses often model workers as possessing an inherent quality score, and weight responses according to the quality of the worker who provided them (Ipeirotis, Provost, and Wang 2010; Karger, Oh, and Shah 2011).

Desired availability of crowd workers. Workers cannot be reliably provisioned on demand like cloud compute resources, and exhibit incredibly high variance in the latency with which they accept tasks posted on marketplaces like MTurk. In addition, for complex work, new workers may require overhead for onboarding and training. Systems where availability is a concern use techniques like retainer pools (Bernstein et al. 2011) variable task pricing (Gao and Parameswaran 2014; Cao et al. 2016), and over-recruitment (Bigham et al. 2010) to ensure that recruitment time doesn't become an issue.

Accommodation of worker preferences. Unfortunately, it is all too easy to design a system that optimizes for traditional system performance metrics without taking into account worker satisfaction. In addition to their skills, workers have individual preferences for different types of work, and will work better when motivated with interesting or meaningful work (Hackman and Oldham 1976; Rogstadius et al. 2011). Whenever possible, system designers should accommodate worker preferences, and avoid optimizations that will improve performance by ignoring preferences.

Technique	Paper Section	System Dynamic Addressed	Crowd Pool Actions Taken
Pool Elasticity	Section 3	Changing task workload	Worker recruitment and release
Pool Stability	Section 4	Worker abandonment	Worker recruitment
Pool Balance	Section 5	Excess idle workers	Inter-pool worker transfer

Table 1: Summary of techniques used by Cioppino.

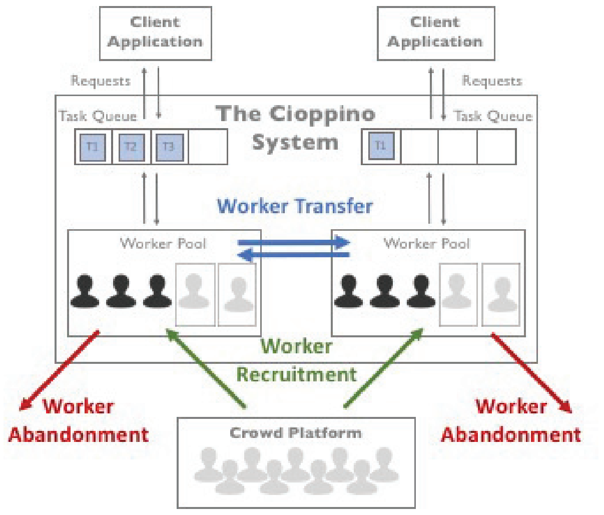


Figure 1: Cioppino architecture diagram.

2.2 Cioppino: a multi-tenant crowd system

In this paper, we address the challenge of managing a multi-tenant crowd. The goal is to design a highly-available system that supports low latency and high throughput for client applications with varying workloads and task types. Taking into account the considerations of Section 2.1, Cioppino is domain-agnostic, uses retainer pools for high availability, is compatible with existing quality control strategies, and introduces several novel techniques (summarized in Table 1): autoscaling crowd pools to improve latency and throughput, explicitly modeling and compensating for worker abandonment, and transferring workers between applications to maximize throughput with consideration for worker preference and training overhead.

Our main insight is that managing crowds of human workers bears a strong resemblance to distributed cluster management. For example, cluster schedulers that support multiple applications must assign each task to a compute node, taking into account heterogeneous hardware and other factors that affect compute latency such as data placement (Delimitrou and Kozyrakis 2014; Venkataraman et al. 2014; Ousterhout et al. 2013). Similarly, Cioppino assigns application tasks to human workers, accounting for varying worker preferences and training. Handling fluctuating task load by autoscaling compute clusters has also received attention. There are a number of approaches to autoscaling (see (Qu, Calheiros, and Buyya 2016) for a recent survey). Techniques that translate well to the crowdsourcing setting include modeling the system with queues or queueing networks (Ali-Eldin, Tordsson, and Elmroth 2012; Jiang et al. 2013) and making decisions to increase or decrease the cluster size using static rules (Netto et al. 2014), currently the state of the art for most cloud providers (Amazon 2016; Google 2017), control theory (Grimaldi et al. 2015), or machine learning (Iqbal, Dailey, and Carrera 2016).

Figure 1 shows the architecture of Cioppino. Client applications issue requests for tasks to be completed. Incoming tasks are enqueued and serviced by workers in a dedicated application crowd worker pool. Cioppino recruits workers from the crowd to perform autoscaling (Section 3) and to compensate for worker abandonment (Section 4). Workers are also transferred between application pools to improve efficiency (Section 5).

2.3 System model

At its core, Cioppino is a queuing system: each type of crowd task (or *client application*) maintains its own queue of tasks, which is serviced by a pool of c_i workers held on retainer. Bernstein et al. (2012) model retainer pools as M/M/c/c queues, in which workers arrive according to a Poisson process and are sent away if there is no demand for new workers. In our setup, we model task arrival, not worker arrival in our queues. Since it is undesirable to discard tasks, we model our system using M/M/C queues (the theory of which is well-established: see Gnedenko and Kovalenko 1989 for an overview). In an M/M/c queue, the arrival rate of tasks is assumed to follow an exponential distribution with parameter λ_i), task completion times are distributed exponentially with parameter μ_i, and incoming tasks queue up (potentially infinitely) if there are no available workers. In practice, since we don't know the true values of λ_i and μ_i, we periodically re-estimate them empirically from the observed task arrival and completion rates in a recent time window.

The value of the queuing model is that it allows us to analyze the expected wait time for a task in the queue. With this information, we will be able to estimate the optimal pool size to service a given workload (Section 3) and estimate the latency impact of transferring workers between application pools (Section 5). Now, we consider the techniques Cioppino uses to manage crowd workers efficiently.

3 Pool Elasticity

In order to optimize its performance, Cioppino must determine a good size for the crowd pool, and adjust that size to meet changes in application workloads, a key consideration since workloads fluctuate dramatically over time (Difallah et al. 2015; Jain et al. 2017). Sizing the worker pool is an important decision. Too small, and its workers will be unable to keep up with the workload, causing a decrease in throughput. Too large, and workers will sit idle, incurring greater cost without any performance benefits. Cioppino leverages its core queueing theory model to identify an optimal queue size, and uses several algorithms for cloud autoscaling to adapt to changing workloads.

3.1 Optimal queue size

In our M/M/c queueing model where tasks are processed by c workers, there is a tradeoff between performance and cost. The larger c is, the lower the expected wait time for an incoming task (leading to system-wide decrease in latency / throughput). However, increasing c also increases the probability of idle workers. Under the retainer pool model, we must pay workers an additional salary to wait if there is no available work, so idle workers have a direct cost.

We can quantify this tradeoff as follows. We write $\rho = \frac{\lambda}{\mu c}$, a measure of the 'traffic intensity' of the system. In an M/M/c queue, the probability that an incoming task must wait is given by

$$\Pi_W = \frac{(c\rho)^c}{c!}\left((1-\rho)\sum_{n=0}^{c-1}\frac{(c\rho)^n}{n!} + \frac{(c\rho)^c}{c!}\right)^{-1}.$$

This is referred to as Erlang's C formula, and can be used to compute both the expected wait time for incoming tasks,

$$\mathbb{E}[W] = \Pi_W \frac{1}{(1-\rho)c\mu} \tag{1}$$

and the expected queue length,

$$\mathbb{E}[L] = \Pi_W \frac{\rho}{1-\rho}.$$

The expected cost of our system is given by the expected number of idle workers times the salary paid for waiting. Following (Haas et al. 2015b), we set the salary to be $s =$ \$0.05/minute, but this can easily be adjusted. So we have that

$$\mathbb{E}[Cost] = s(\max\{0, c - \mathbb{E}[L]\}). \tag{2}$$

To find an optimal c^*, we must relate Equations 1 and 2. We explicitly trade off cost and performance by minimizing a weighted average of the two: $c^* = \arg\min_c \eta\mathbb{E}[W] + (1-\eta)\mathbb{E}[Cost]$, where η is a system parameter establishing a preference for one or the other. This equation has no clean closed form, but since it is convex, we can solve numerically for the exact optimum c^*.

3.2 Autoscaling algorithms

For an application i, the optimal queue size c_i^* that we derived in Section 3.1 depends on the parameters λ_i, the rate of incoming new tasks, and μ_i, the rate at which individual tasks are completed. While we expect μ_i to remain fairly stable over time for a specific task type, λ_i varies with the application workload, which is not guaranteed to be constant. Our empirical estimate of λ_i is intentionally calculated over a recent window of time to take this into account, and every time we observe a change in λ_i, we must re-size the pool to compensate. Inspired by the literature on cloud autoscaling, we implement two approaches to autoscaling in Cioppino: rule-based autoscaling and control-theoretic autoscaling.

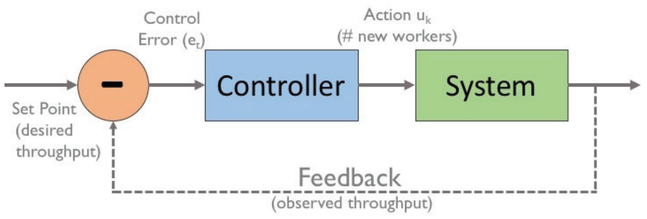


Figure 2: Control system block diagram.

Rule-based autoscaling. Most cloud IaaS providers (e.g., EC2 (Amazon 2016), GCE (Google 2017), etc.) currently offer threshold-based policies for autoscaling application hardware, for example, *"add 2 VMs to my application if the average CPU utilization among current VMs rises above 80%"*. This approach is attractive because it is easy to deploy and understand, but writing effective rules requires deep understanding of the application workload. In Cioppino, we allow rules to be written based on system performance metrics (e.g., queue length, observed throughput, observed latency, pool size) or queueing model metrics (e.g., λ_i, μ_i, or our optimal estimate for c_i).

All rules are composed of a *condition* that triggers the rule and an *action* to take if the rule is triggered. A condition relates a parameter to a threshold, and may contain multiple clauses connected by boolean operators. For example, the following rule recruits a new worker whenever the queue builds up or the pool becomes smaller than the estimated optimum:

```
if $L_i > 10$ or $c_i^* - c_i > 5$
then action("recruit", 1).
```

Rules that rely on c_i^* have particular promise, since they retain the ease-of-use of metric-based rules, but rely on the system model instead of requiring administrators to understand application workload.

Control-theoretic autoscaling. Systems for cloud autoscaling use a variety of control-theoretic models to compute the number of additional VMs that need provisioning, including fixed-gain controllers, adaptive controllers, and fuzzy models (Lorido-Botran, Miguel-Alonso, and Lozano 2014). Since to our knowledge this work represents the first attempt at autoscaling crowd pools, we limit ourselves to fixed-gain PID controllers, which use a simple linear model to relate the control action to the observed error in the *process variable* or parameter of interest. Figure 2 shows a block diagram of our closed-loop control system. At each iteration of the loop, the user provides a target value (set point) for the process variable. The controller computes the desired change in the number of crowd workers, then takes action to recruit or release workers. In a PID controller, the desired change in the number of workers is computed as

$$u_t = K_p e_t + K_i \sum_{j=0}^{t} e_j + K_d(e_t - e_{t-1}).$$

The first (Proportional) term reflects the current gap between the set point and the observed value of the process variable

(e_t). The second (Integral) term reflects the history of observed errors. The third (Derivative) term reflects the current rate of change in error. The relative magnitude of these terms are controlled by the three weight parameters K_p, K_i, and K_d. We tune these parameters manually, though techniques for offline and online autotuning of PID systems has received considerable attention in the literature. In our experiments, we evaluate PID controllers built on two process variables: system throughput, for which e_t is the number of arrived tasks minus the number of completed tasks at timestep t; and pool size, for which e_t is the optimum c_i^* minus the pool size c_i at timestep t.

4 Pool Stability

There are a number of challenges to running crowd retainer pools in production systems. One crucial limitation is that of worker abandonment.

4.1 Worker abandonment

In an idealized model of retainer pools, the requester hires a pool of workers, employs them until the desired work is complete, then pays and releases them. In reality, especially in online crowd marketplaces, workers abandon tasks without warning for a variety of reasons—fatigue, boredom, accidentally closing browser tabs, etc. For example, (Bernstein et al. 2011) found that about 10-20% of workers abandoned the pool before accepting their next task, and analysis of the CLAMShell trace described in Section 6.1 shows that workers on average only remain in the pool for 5-6 minutes. As a result, a production deployment of crowd worker pools must account for abandonment by constantly recruiting new workers to the pool. Cioppino introduces a novel mechanism to do so called *pool stability*.

4.2 Pool stability

Though it would be nice to be able to simply recruit another worker every time one leaves, the delay associated with hiring new workers might leave the pool short-handed for a significant period of time. Here, we examine three recruitment strategies to mitigate this effect: rule-based, average-rate, and hybrid. These approaches are evaluated in Section 6.2.

Rule-based recruitment. As with autoscaling techniques, rules can be explicitly defined to respond to worker abandonment. Cioppino permits rules of the form:

if $c_i < thresh$, then action("recruit", n).

Rules can also be defined in terms of the rate of abandonment. As before, rules are easy to define and apply, but hard to tune without knowledge of the dynamics of abandonment.

Average-rate recruitment. Rule-based approaches might perform poorly if the rule condition is poorly tuned to the abandonment rate. Average-rate recruitment seeks to avoid this problem. It continually maintains a count n_{window} of the number of workers who abandoned the pool in the last t_{window} seconds, then recruits $\hat{a}_i = \frac{n_{window}}{t_{window}}$ workers every second. If $\hat{a}_i$ is non-integral, then the algorithm recruits $\lfloor \frac{n_{window}}{t_{window}} \rfloor$ workers, and with probability $\frac{n_{window} \mod t_{window}}{t_{window}}$ recruits an additional worker. In expectation, this achieves the correct rate.

Hybrid recruitment. Average-rate recruitment avoids overreacting to spikes in abandonment by recruiting at a constant rate based on the empirical observed average. However, it ignores the current size of the pool, and cannot react if the pool size grows too large due to over-recruitment. Hybrid recruitment seeks to achieve the best of both worlds. It runs the average-rate recruitment algorithm, but additionally uses a threshold-based rule to prevent recruitment if the pool is too large. That is, it can be summarized with the rule:

if $c_i < thresh$, then action("recruit", avg_rate_recruitment(a_{window}, n_{window})).

5 Pool Balance

Autoscaling each application's pool individually will successfully adjust to changing workloads, but it may result in inefficiency because it doesn't take into account the scaling needs of other pools. Cioppino uses a novel algorithm called *Pool Balance* to determine when it is efficient to shift workers between applications, taking into account two subtleties: first, there is a training overhead associated with assigning a worker a task type they've never done before; and second, worker preferences should be taken into account when performing transfers.

Algorithm 1 Pool Balance.

```
best_tset_score = $-\infty$, best_tset = None
for tset in generate_candidate_tsets():
    completion_times = {}
    preferences = {}
    for pool in All_Pools:
        new_pool = simulate_transfer(tset, pool)
        completion_times += completion_est(tset, new_pool)
        preferences += net_preference(new_pool)
    perf_score = perf_ts(completion_times)
    h_score = h_ts(preferences)
    net_score = $\omega *$ h_score + $(1 - \omega) *$ perf_score
    if net_score $>$ best_tset_score:
        best_tset = transfer_set
        best_tset_score = net_score
apply_tset(best_tset)
```

5.1 Approach overview

Algorithm 1 describes our algorithm for choosing a set of idle workers to transfer to other pools (a *transfer set*, consisting of zero or more (worker, destination-pool) pairs). For each transfer set we consider, we simulate the transfers, then estimate how it will affect the processing of currently backed up tasks and the preferences of workers for their current work. Because the space of possible transfer sets is large, we use heuristics to limit the candidate transfer sets we consider. Our algorithm runs in a loop during system execution: periodically, it generates candidate transfer sets, evaluates them, and applies the best one.

5.2 Estimating queue completion time

The function `completion_est` in Algorithm 1 estimates the time it will take for an individual queue to empty given its current length L_i and its current workforce c_i. This estimate is equivalent to the expected wait time of the next item to arrive in the queue, $\mathbb{E}[W_i|L_i]$. To compute the estimated waiting time, we observe that the task will wait in line until L_i items have been processed and exited the queue. Let D_k be the kth interdeparture time since the arrival of the last task in the queue. Then $\mathbb{E}[W_i] = \mathbb{E}[\sum_{k=1}^{L_i} D_k]$. Clearly, the D_k are independently exponentially distributed with mean $\frac{1}{c_i\mu_i}$. The sum of n independent exponentially distributed random variables with identical mean μ is Erlang-distributed with shape parameter n and rate μ, so we have that:

$$\mathbb{E}[W_i] = \mathbb{E}\left[\sum_{k=1}^{L_i} D_k\right] = \mathbb{E}[Er(x; L_i, c_i\mu_i)] = \frac{L_i}{c_i\mu_i}. \quad (3)$$

When we transfer workers between pools, however, workers must learn to do the new task type. Thus, we adjust Equation 3 to account for training time, making one conservative assumption to simplify the analysis: that new workers aren't added to the pool until all workers have completed training. A transfer thus breaks queue processing time into two phases: during training, the queue can't make use of new workers, but once training is done, it can. Let us assume that training times, like task times, are exponentially distributed with parameter tr_i, let k_i^{TS} be the number new workers in pool i under transfer set TS, and let c_i^{TS} be the total workers after TS is applied: $c_i + k_i^{TS} = c_i^{TS}$.

First, we analyze the expected length of the training period. Training ends when all workers have finished training, so (using that the expectation of the maximum of n independent exponential random variables with parameter λ is $\frac{1}{\lambda}\sum_{j=1}^{n}\frac{1}{j}$ — see (Lugo 2011) for a proof):

$$\mathbb{E}[t_{train}] = \mathbb{E}[\max(tr_{i,1}, \ldots, tr_{i,k_i^{TS}})] = \frac{1}{tr_i}\sum_{j=1}^{k_i^{TS}}\frac{1}{j}.$$

During training, the queue processes tasks at an average rate of $\mu_i(c_i^{TS} - k_i^{TS})$, so an expected $\frac{\mu_i(c_i^{TS}-k_i^{TS})}{tr_i}\sum_{j=1}^{k_i^{TS}}\frac{1}{j}$ tasks exit the queue during training. Call this quantity L_{tr}. Now we can express the adjusted $\mathbb{E}[W_i]$ in terms of the training and post-training phases:

$$\mathbb{E}[W_i^{TS}] = \begin{cases} \frac{L_i}{(c_i^{TS}-k_i^{TS})\mu_i} & \text{if } L_i \le L_{tr}, \\ \mathbb{E}[t_{train}] + \frac{L_i - L_{tr}}{c_i^{TS}\mu_i} & \text{otherwise.} \end{cases} \quad (4)$$

That is, if the queue is small enough to be completely processed during training, then the expected processing time is as if the new workers didn't exist. If not, then the expected processing time is the expected training time plus the time to process the tasks that weren't processed during training, using the newly trained workers.

5.3 Evaluating candidate transfer sets

Performance. Now that we have an estimate for the expected time to process an individual queue, we can evaluate the performance utility of a transfer set TS. First, we compute the set $\mathcal{W} = \{\mathbb{E}[W_i^{TS}] : 0 \le i < |Apps|\}$ using Equation 4. Then, we compute the performance score $Perf^{TS}$ (the `perf_ts` function in Algorithm 1) by evaluating the utility of this set. In our experiments, we use $Perf^{TS} = 1/\text{median}(\mathcal{W})$, though other utility functions could be used based on system performance goals.

Worker preferences. $Perf^{TS}$ only accounts for the performance effects of a transfer set. However, transferring a worker may be undesirable if the worker doesn't enjoy the task type of the destination application, and two equally efficient transfer sets might not be equally sensitive to workers' task type preferences. To model this, let $p_{i,j} \in [0,1]$ represent the jth worker's preference for the task type of application i. In simulation, we generate the $p_{i,j}$ uniformly at random, but in a live system new workers could establish preferences for the available task types when they join. To evaluate a transfer set TS, in addition to computing $\mathbb{E}[W_i^{TS}]$, we compute the *net preference* of the transfer set,

$$H^{TS} = \sum_{i \in Apps}\sum_{j \in Apps_i^{TS}} p_{i,j}. \quad (5)$$

Then, we update our transfer set scoring to take the weighted average of the two metrics:

$$Score^{TS} = \omega * H^{TS} + (1-\omega) * Perf^{TS}, \quad (6)$$

where ω is a system parameter that sets the relative importance of worker satisfaction to system performance. We evaluate our techniques for their effect on net preference in Section 6.2.

5.4 Searching the transfer set space

Unfortunately, the space of candidate transfer sets is large, making a brute-force comparison of all possible transfer sets impractical. To see this, observe that we could transfer any subset of idle workers in the system to any other app in the system, so if there are A apps in the system and $C = \sum_{i=0}^{A-1} c_i$ total workers, there are up to $\sum_{j=0}^{C}\binom{C}{j}A^j = (A+1)^C$ possible transfer sets.

Instead, we apply heuristics to cut down the space we search over. First, we don't consider transfer sets that send workers to applications having $L_i = 0$, since those applications are unlikely to need additional workers. Note that this rules out transfer sets that swap workers symmetrically between pools, since all applications with idle workers have $L_i = 0$. Additionally, we cap the size of a transfer set at 10. Finally, instead of considering all A^k transfer sets for a worker subset of size k, we use a greedy heuristic to generate a single assignment of workers to destinations. The heuristic works as follows: for each worker pool, we compute $\mathbb{E}[W_i]$ from Equation 3 and H^{TS} from Equation 5. Then we assign each worker in the subset to the destination pool with the largest $\omega * H^{TS} + (1-\omega) * \mathbb{E}[W_i]$, incrementing c_i and recomputing the score after each assignment.

6 Evaluation

In this section, we evaluate Cioppino's techniques in isolation and together, demonstrating simultaneously a 20% improvement in throughput and a 19× decrease in cost. For space reasons, we present only a summary of results here. A detailed evaluation can be found in Appendix A (submitted as supplemental material).

6.1 Experimental Setup

We evaluated Cioppino's techniques on a multi-tenant crowd pool simulator implemented in python. The experiments were conducted on an Amazon EC2 c4.4xlarge instance with 16 vCPUs and 30 GB of RAM. Workload and crowd behavioral data were either generated or trace-based.

Generated data. In some experiments, crowd worker behavior is simulated. Abandonment rates are modeled by a parameter α: each worker chooses to abandon the system with probability α at the end of their task. Recruitment time is modeled as a Poisson process with rate r. Worker preferences are generated uniformly at random in $[0, 1)$ for each task type (client application) in the system, then normalized to sum to 1. In addition, workload data are simulated, including the task completion rate μ_i, the arrival rate of tasks λ_i, and initial pool sizes c_i for client applications.

Trace-based data. Some experiments are trace-based simulations. The trace was extracted from a live deployment of low-latency retainer pools for simple labeling tasks on MTurk described in (Haas et al. 2015b).[1] It consists of three streams of data in chronological order: the worker recruitment times for all new workers who joined the pool (n=6126 recruitments, mean time 91.3 seconds, standard deviation 113.9 seconds), the task duration for all tasks processed in the pool (n=6725 tasks, mean time 1.97 seconds, standard deviation 0.87 seconds), and the duration that workers remained in the pool before abandoning it (n=2999 abandonments, mean time 316.6 seconds, standard deviation 211.1 seconds).

6.2 Individual Techniques

Pool elasticity. We compare three pool elasticity techniques against the static baseline (S) that begins with the optimal pool size for the entire workload and does no autoscaling. Rule-based autoscaling (RB) uses two rules: "`if` $L_i > 10$ `or` $c_i^* - c_i > 5$ `then action('recruit', 1)`", and "`if` $c_i^* - c_i < -5$ `then action('release', 1)`." This rule was hand-tuned to perform well against our workloads to illustrate the potential of rule-based techniques, but because rule design requires intimate knowledge of the workload, other workloads might require different rules to perform well. Control-theoretic autoscaling uses the PID controller described in Section 3.2, and is evaluated with two different process variables: system throughput (CT-t) and pool size (CT-p).

Qualitatively, the control-theoretic techniques converge most quickly to the new optimal pool size after a workload change. Figure 3a shows the cost and throughput achieved by the techniques after an increase in workload. If cost is a factor, CT-t is the obvious best choice for pool autoscaling (5-8× cheaper than the other techniques and 88× cheaper than the baseline S). However, CT-p consistently achieves the best throughput (2-12% higher than CT-t, 12% higher than RB, and 17% higher than S) by keeping the worker pool at the optimal size prescribed by the Cioppino queuing model.

Pool stability. We compare three pool stability techniques against the null technique (N) that never recruits workers to compensate for worker abandonment. Rule-based stability (RB) recruits new workers whenever the pool size dips too low, implementing the rule: "`if` $c_i^* - c_i >$ 5, `then action('recruit', 1)`". Average-rate recruitment (AR) recruits new workers at a constant average rate. Hybrid recruitment (HR) uses the same rule as RB combined with the rate of AR, as described in Section 4.2.

Qualitatively, RB does not recruit quickly enough, keeping the pool at a constant but slightly too low level, whereas AR over-recruits slightly. As desired, HR strikes a balance between the two. Figure 3b illustrates the performance of the techniques with abandonment rate $\alpha = 0.1$ and recruitment rate $r = 0.02$. AR, the strategy that recruits most aggressively, shows the strongest performance, ranging from 5-15% higher throughput than HR, 14% - 2× higher throughput than RB, and up to 14× higher throughput than N. However, because it over-recruits, it incurs significant cost: 3.3× HR, and 47× RB.

Pool balance. We compare two pool balance techniques against the null technique (NT) that never transfers workers between pools. (PB) is the pool balance algorithm described in Section 5. Random transferring (RT) sends idle workers to a random pool.

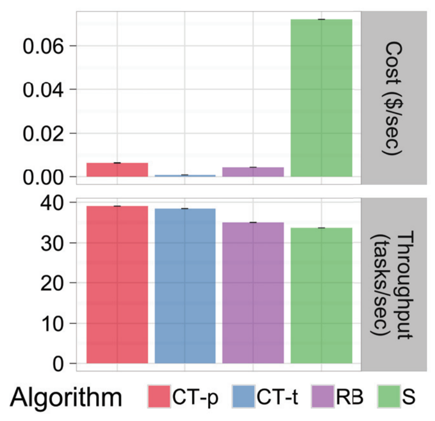
Qualitatively, though RT successfully rebalances workers to arrive at the optimal pool size, it does so much more slowly than PB. As a result, PB consistently achieves a higher throughput (18% higher than RT and 76% higher than NT) at less cost (10× less than RT and 27× less than NT). PB rebalances workers based on their preferences, so its workers exhibit a 43% higher preference for the application they are ultimately transferred to than workers in the RT and NT conditions. Figure 3c demonstrates these trends.

6.3 End-to-end evaluation

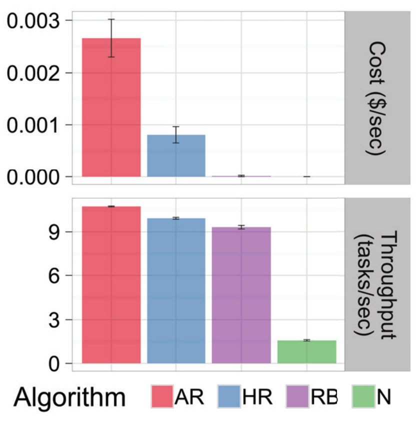
To evaluate the overall efficiency of Cioppino, we compare two scenarios, a state-of-the-art baseline and a combination of the best performing techniques of Cioppino. The baseline behaves like typical crowd systems today: it does not autoscale its crowd pool (pool elasticity policy S), it uses rule-based recruitment to respond to worker abandonment (pool stability policy RB), and it does not do any inter-pool worker training and transfer (pool balance policy NT). The Cioppino system uses throughput-based control-theoretic autoscaling (CT-t), hybrid recruitment for pool stability (HR), and our novel pool balancing algorithm (PB).

Figure 4 illustrates the systems' performance on the trace described in Section 6.1. Qualitatively, the Cioppino system is much better at keeping the pool size close to the optimal value in spite of changing workloads (though there is increased noise now that our data is real, not generated). Quantitatively, Cioppino dominates Baseline on all metrics: it has a 20% higher throughput, 19.4× lower cost, and 2.3× higher net worker preference.

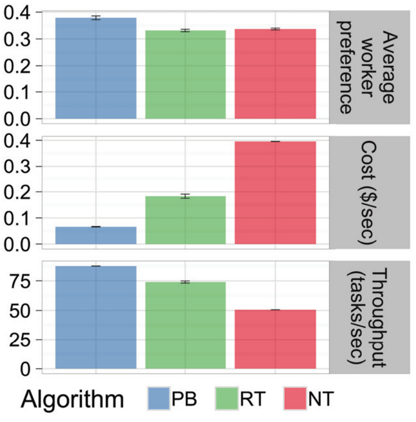
[1]Trace data and additional description can be found at http://thisisdhaas.com/clamshell_trace.html.



(a) Pool Elasticity



(b) Pool Stability



(c) Pool Balance

Figure 3: Summary of results from evaluating individual techniques.

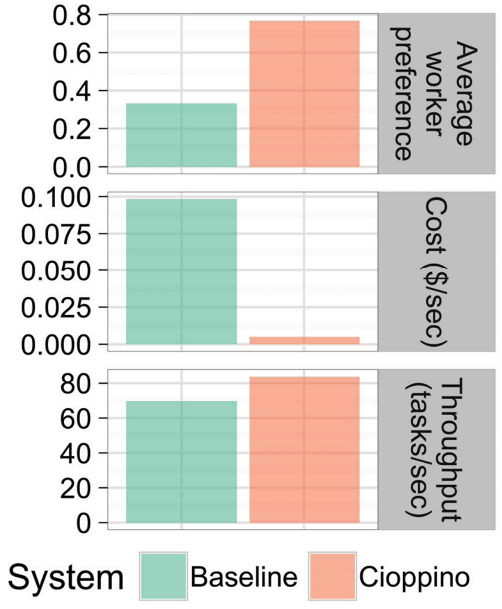


Figure 4: End to end system comparison on the crowd trace.

7 Conclusion & Future Directions

In this work, we have introduced Cioppino, a system for holistic multi-tenant crowd management that reduces cost and increases efficiency while accounting for human factors that affect system performance. Cioppino models crowd task processing as a queueing system, and introduces novel algorithms for autoscaling, compensating for worker abandonment, and rebalancing workers across application pools. Multi-tenant crowd systems should incorporate these techniques to make the most of their crowdsourced workforce.

Cioppino represents a single point on the spectrum between systems that treat humans as programming abstractions and those that model human behavior exhaustively. In addition to improving the techniques described in this paper (for example, by exploring machine-learning or time-series-analysis based autoscaling techniques), future work in the design of performant crowd systems should investigate the spectrum itself. We would like to investigate when it is necessary to model individual worker behaviors in order to guarantee performance, and experiment with dynamically generating system models appropriate to the needs of the workload being run and the crowd running it.

We are grateful to the hundreds of crowd workers who made this work possible. This research is supported in part by NSF CISE Expeditions Award CCF-1139158, DOE Award SN10040 DE-SC0012463, and DARPA XData Award FA8750-12-2-0331, and gifts from Amazon Web Services, Google, IBM, SAP, The Thomas and Stacey Siebel Foundation, Adatao, Adobe, Apple, Inc., Blue Goji, Bosch, Cisco, Cray, Cloudera, EMC2, Ericsson, Facebook, Fujitsu, Guavus, HP, Huawei, Informatica, Intel, Microsoft, NetApp, Pivotal, Samsung, Schlumberger, Splunk, Virdata and VMware.

References

Ali-Eldin, A.; Tordsson, J.; and Elmroth, E. 2012. An adaptive hybrid elasticity controller for cloud infrastructures. In *NOMS*, 204–212. IEEE.

Amazon. 2016. Amazon auto scaling service. http://aws.amazon.com/autoscaling/.

Bernstein, M. S.; Little, G.; Miller, R. C.; et al. 2010. Soylent: a word processor with a crowd inside. *UIST*.

Bernstein, M. S.; Brandt, J.; Miller, R. C.; and Karger, D. R. 2011. Crowds in two seconds: enabling realtime crowd-powered interfaces. *UIST*.

Bernstein, M. S.; Karger, D. R.; Miller, R. C.; and Brandt, J. 2012. Analytic Methods for Optimizing Realtime Crowdsourcing. *Collective Intelligence*.

Bigham, J. P.; Jayant, C.; Ji, H.; et al. 2010. *VizWiz: nearly real-time answers to visual questions*. UIST.

Callison-Burch, C. 2009. Fast, cheap, and creative: evaluating translation quality using Amazon's Mechanical Turk. *EMNLP*.

Cao, C. C.; Liu, Z.; Chen, L.; and Jagadish, H. V. 2016. Tuning crowdsourced human computation. *CoRR* abs/1610.04429.

Chen, K.-T.; Wu, C.-C.; Chang, Y.-C.; and Lei, C.-L. 2009. A crowdsourceable qoe evaluation framework for multimedia content. In *Proceedings of the 17th ACM International Conference on Multimedia*, MM '09, 491–500. ACM.

Cheng, J., and Bernstein, M. S. 2015. Flock: Hybrid crowd-machine learning classifiers. In *Proceedings of the 18th ACM Conference on Computer Supported Cooperative Work & Social Computing*, CSCW '15, 600–611.

Dean, J., and Ghemawat, S. 2008. MapReduce: simplified data processing on large clusters. *Communications of the ACM*.

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

Difallah, D. E.; Catasta, M.; Demartini, G.; Ipeirotis, P. G.; and Cudré-Mauroux, P. 2015. The dynamics of micro-task crowdsourcing: The case of amazon mturk. In *Proceedings of the 24th International Conference on World Wide Web*, WWW '15, 238–247.

Difallah, D. E.; Demartini, G.; and Cudré-Mauroux, P. 2016. Scheduling human intelligence tasks in multi-tenant crowd-powered systems. In *Proceedings of the 25th International Conference on World Wide Web*, WWW '16, 855–865. International World Wide Web Conferences Steering Committee.

Gao, Y., and Parameswaran, A. 2014. Finish them!: Pricing algorithms for human computation. *Proc. VLDB* 7(14):1965–1976.

Ghodsi, A.; Zaharia, M.; Hindman, B.; Konwinski, A.; Shenker, S.; and Stoica, I. 2011. Dominant resource fairness: Fair allocation of multiple resource types. In *Proceedings of the 8th USENIX Conference on Networked Systems Design and Implementation*, NSDI'11, 323–336. Berkeley, CA, USA: USENIX Association.

Gnedenko, B. V., and Kovalenko, I. N. 1989. *Introduction to Queueing Theory (2Nd Ed)*. Cambridge, MA, USA: Birkhauser Boston Inc.

Gokhale, C.; Das, S.; Doan, A.; et al. 2014. Corleone: hands-off crowdsourcing for entity matching. *SIGMOD*.

Google. 2017. Autoscaling groups of instances. https://cloud.google.com/compute/docs/autoscaler/.

Grimaldi, D.; Persico, V.; Pescape, A.; Salvi, A.; and Santini, S. 2015. A feedback-control approach for resource management in public clouds. In *2015 IEEE Global Communications Conference (GLOBECOM)*, 1–7.

Haas, D.; Ansel, J.; Gu, L.; and Marcus, A. 2015a. Argonaut: Macrotask crowdsourcing for complex data processing. *Proc. VLDB* 8(12):1642–1653.

Haas, D.; Wang, J.; Wu, E.; and Franklin, M. J. 2015b. Clamshell: Speeding up crowds for low-latency data labeling. *Proc. VLDB* 9(4):372–383.

Hackman, J., and Oldham, G. R. 1976. Motivation through the design of work: test of a theory. *Organizational Behavior and Human Performance* 16(2):250 – 279.

Ipeirotis, P. G.; Provost, F.; and Wang, J. 2010. Quality management on Amazon Mechanical Turk. *SIGKDD*.

Ipeirotis, P. G 2010. Analyzing the Amazon Mechanical Turk marketplace. *ACM Crossroads*.

Iqbal, W.; Dailey, M. N.; and Carrera, D. 2016. Unsupervised learning of dynamic resource provisioning policies for cloud-hosted multitier web applications. *IEEE Systems Journal* 10(4):1435–1446.

Jain, A.; Sarma, A. D.; Parameswaran, A.; and Widom, J. 2017. Understanding workers, developing effective tasks, and enhancing marketplace dynamics: A study of a large crowdsourcing marketplace. *Proc. VLDB Endow.* 10(7):829–840.

Jiang, J.; Lu, J.; Zhang, G.; and Long, G. 2013. Optimal cloud resource auto-scaling for web applications. In *2013 13th IEEE/ACM International Symposium on Cluster, Cloud, and Grid Computing*.

Karger, D. R.; Oh, S.; and Shah, D. 2011. Iterative Learning for Reliable Crowdsourcing Systems. *Advances in neural information processing systems (NIPS)*.

Kittur, A.; Smus, B.; Khamkar, S.; and Kraut, R. E. 2011. Crowdforge: Crowdsourcing complex work. In *Proceedings of UIST '11*, 43–52.

Krishna, R. A.; Hata, K.; Chen, S.; Kravitz, J.; Shamma, D. A.; Fei-Fei, L.; and Bernstein, M. S. 2016. Embracing error to enable rapid crowdsourcing. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, CHI '16, 3167–3179.

Kulkarni, A.; Narula, P.; Rolnitzky, D.; and Kontny, N. 2014. Wish: Amplifying creative ability with expert crowds. *HCOMP*.

Little, G. 2009. Turkit: Tools for iterative tasks on mechanical turk. In *2009 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, 252–253.

Lorido-Botran, T.; Miguel-Alonso, J.; and Lozano, J. A. 2014. A review of auto-scaling techniques for elastic applications in cloud environments. *J. Grid Comput.* 12(4):559–592.

Lugo, M. 2011. The expectation of the maximum of exponentials. http://www.stat.berkeley.edu/~mlugo/stat134-f11/exponential-maximum.pdf.

Marcus, A., and Parameswaran, A. 2015. Crowdsourced data management industry and academic perspectives. *Foundations and Trends in Databases*.

Marcus, A.; Karger, D.; Madden, S.; Miller, R.; and Oh, S. 2012. Counting with the crowd. *VLDB*.

Mozafari, B.; Sarkar, P.; Franklin, M.; et al. 2014. Scaling up crowd-sourcing to very large datasets: a case for active learning. *VLDB*.

Netto, M. A.; Cardonha, C.; Cunha, R. L.; and Assunçao, M. D. 2014. Evaluating auto-scaling strategies for cloud computing environments. In *Modelling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2014 IEEE 22nd International Symposium on*, 187–196. IEEE.

Ousterhout, K.; Wendell, P.; Zaharia, M.; and Stoica, I. 2013. Sparrow: Distributed, low latency scheduling. In *Proceedings of the Twenty-Fourth ACM Symposium on Operating Systems Principles*, SOSP '13, 69–84. New York, NY, USA: ACM.

Qu, C.; Calheiros, R. N.; and Buyya, R. 2016. Auto-scaling web applications in clouds: A taxonomy and survey. *CoRR* abs/1609.09224.

Ramesh, A.; Parameswaran, A.; Garcia-Molina, H.; and Polyzotis, N. 2012. Identifying Reliable Workers Swiftly. Technical report, Stanford University.

Retelny, D.; Robaszkiewicz, S.; To, A.; et al. 2014. Expert crowdsourcing with flash teams. In *Proceedings of UIST '14*, 75–85.

Rogstadius, J.; Kostakos, V.; Kittur, A.; Smus, B.; Laredo, J.; and Vukovic, M. 2011. An assessment of intrinsic and extrinsic motivation on task performance in crowdsourcing markets. In *ICWSM*.

Rzeszotarski, J. M.; Chi, E.; Paritosh, P.; and Dai, P. 2013. Inserting micro-breaks into crowdsourcing workflows. In *First AAAI Conference on Human Computation and Crowdsourcing*.

Sheng, V. S.; Provost, F.; and Ipeirotis, P. G. 2008. Get another label? improving data quality and data mining using multiple, noisy labelers. In *Proceedings of SIGKDD '08*, 614–622.

Venkataraman, S.; Panda, A.; Ananthanarayanan, G.; Franklin, M. J.; and Stoica, I. 2014. The power of choice in data-aware cluster scheduling. In *Proceedings of OSDI '14*, 301–316.

Wang, J.; Kraska, T.; Franklin, M. J.; and Feng, J. 2012. CrowdER: Crowdsourcing Entity Resolution. *VLDB*.