Cocktail: A Multidimensional Optimization for Model Serving in Cloud

Jashwant Raj Gunasekaran, Cyan Subhra Mishra, Prashanth Thinakaran, Bikash Sharma,
Mahmut Taylan Kandemir, Chita R. Das
The Pennsylvania State University, University Park, PA
{jashwant, cyan, prashanth, mtk2, cxd12}@psu.edu, bikash.nitrkl@acm.org

Abstract

With a growing demand for adopting ML models for a variety of application services, it is vital that the frameworks serving these models are capable of delivering highly accurate predictions with minimal latency along with reduced deployment costs in a public cloud environment. Despite high latency, prior works in this domain are crucially limited by the accuracy offered by individual models. Intuitively, model ensembling can address the accuracy gap by intelligently combining different models in parallel. However, selecting the appropriate models dynamically at runtime to meet the desired accuracy with low latency at minimal deployment cost is a nontrivial problem. Towards this, we propose *Cocktail*, a cost effective ensembling-based model serving framework. *Cocktail* comprises of two key components: (i) a dynamic model selection framework, which reduces the number of models in the ensemble, while satisfying the accuracy and latency requirements; (ii) an adaptive resource management (RM) framework that employs a distributed proactive autoscaling policy, to efficiently allocate resources for the models. The RM framework leverages transient virtual machine (VM) instances to reduce the deployment cost in a public cloud. A prototype implementation of *Cocktail* on the AWS EC2 platform and exhaustive evaluations using a variety of workloads demonstrate that *Cocktail* can reduce deployment cost by 1.45×, while providing 2× reduction in latency and satisfying the target accuracy for up to 96% of the requests, when compared to state-of-the-art model-serving frameworks.

1 Introduction

Machine Learning (ML) has revolutionized user experience in various cloud-based application domains such as product recommendations [70], personalized advertisements [44], and computer vision [13, 43]. For instance, Facebook [44, 82] serves trillions of inference requests for user-interactive applications like ranking new-feeds, classifying photos, etc. It is imperative for these applications to deliver accurate predictions at sub-millisecond latencies [27,34,35,39,44,83] as they critically impact the user experience. This trend is expected to perpetuate as a number of applications adopt a variety of ML models to augment their services. These ML models are typically trained and hosted on cloud platforms as service endpoints, also known as *model-serving* framework [6, 28, 60]. From the myriad of ML flavours, Deep Neural Networks (DNNs) [54] due to their multi-faceted nature, and highly generalized and accurate learning patterns [45,73] are dominating the landscape by making these model-serving frameworks accessible to developers. However, their high variance due to the fluctuations in training data along with compute and memory intensiveness [59, 65, 84] has been a major impediment in designing models with high accuracy and low latency. Prior model-serving frameworks like InFaas [83] are confined by the accuracy and latency offered by such individual models.

Unlike single-model inferences, more sophisticated techniques like *ensemble learning* [15] have been instrumental in allowing model-serving to further improve accuracy with multiple models. For example, by using the ensembling [1] technique, images can be classified using multiple models *in parallel* and results can be combined to give a final prediction. This significantly boosts accuracy compared to single-models, and for this obvious advantage, frameworks like Clipper [27] leverage ensembling techniques. Nevertheless, with ensembling, the very high resource footprint due to sheer number of models that need to be run for each request [27, 56], exacerbates the public cloud deployment costs, as well as leads to high variation in latencies. Since cost plays a crucial role in application-provider consideration, it is quintessential to minimize the deployment costs, while maximizing accuracy with low latency. Hence, the non-trivial challenge here lies in making the cost of ensembling predictions analogous to single model predictions, while satisfying these requirements.

Studying the state-of-the-art ensemble model-serving frameworks, we observe the following critical shortcomings:

• Ensemble model selection policies used in frameworks like Clipper [27] are static, as they *ensemble all available models* and focus solely on minimizing loss in accuracy. This leads to higher latencies and further inflates the resource footprint, thereby accentuating the deployment costs.

• Existing ensemble weight estimation [87] has *high computational complexity* and in practice is limited to a small set of off-the-shelf models. This leads to significant loss in accuracy. Besides, employing linear ensembling techniques such as model averaging are compute intensive [80] and not scalable for a large number of available models.

• Ensemble systems [27, 80] are *not focused towards model deployment* in a public cloud infrastructure, where resource

[1] We refer to ensemble-learning as ensembling throughout the paper.

selection and procurement play a pivotal role in minimizing the latency and deployment costs. Further, the resource provisioning strategies employed in single model-serving systems are *not directly extendable* to ensemble systems.

These shortcomings collectively motivate the central premise of this work: *how to solve the complex optimization problem of cost, accuracy and latency for an ensembling framework?* In this paper, we present and evaluate *Cocktail*[2], which to our knowledge is the first work that proposes a cost-effective model-serving system by exploiting ensembling techniques for classification-based inference, to deliver high accuracy and low latency predictions. *Cocktail* adopts a three-pronged approach to solve the optimization problem. First, it uses a dynamic model selection policy to significantly reduce the number of models used in an ensemble, while meeting the latency and accuracy requirements. Second, it utilizes distributed autoscaling policies to reduce the latency variability and resource consumption of hosting ensemble models. Third, it minimizes the cost of deploying ensembles in a public cloud by taking advantage of transient VMs, as they can be 70-90% cheaper [3] than traditional VMs. *Cocktail*, by coalescing these benefits, is capable of operating in a region of optimal cost, accuracy and latency (shown in Figure 1) that prior works cannot achieve. Towards this, the **key contributions** of the paper are summarized below:

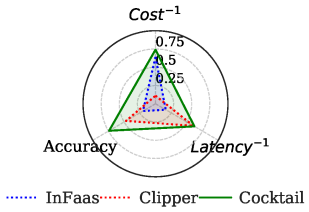


Figure 1: Benefits of *Cocktail*. Results are normalized (higher the better).

1. By characterizing accuracy *vs.* latency of ensemble models, we identify that prudently selecting a subset of available models under a given latency can achieve the target accuracy. We leverage this in *Cocktail*, to design a novel dynamic model selection policy, which ensures accuracy with significantly reduced number of models.
2. Focusing on classification-based inferences, it is important to minimize the bias in predictions resulting from multiple models. In *Cocktail*, we employ a per-class weighted majority voting policy, that makes it scalable and effectively breaks ties when compared to traditional weighted averaging, thereby minimizing the accuracy loss.
3. We show that uniformly scaling resources for all models in the ensemble leads to over-provisioning of resources and towards minimizing it, we build a distributed weighted auto-scaling policy that utilizes the *importance sampling* technique to proactively allocate resources to every model. Further, *Cocktail* leverages transient VMs as they are cheaper, to drastically minimize the cost for hosting model-serving infrastructure in a public cloud.

[2]Cocktail is ascribed to having the perfect blend of models in an ensemble.

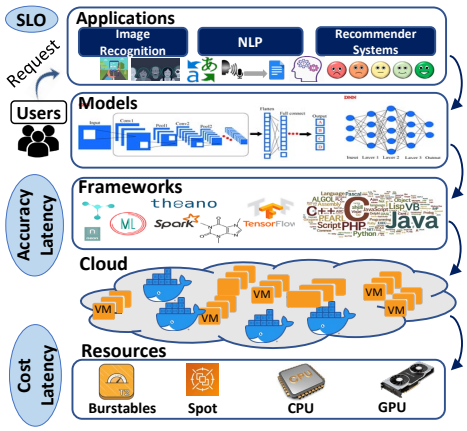


Figure 2: The overall framework for model-serving in public cloud.

4. We implement a prototype of *Cocktail* using both CPU and GPU instances on AWS EC2 [5] platform and extensively evaluate it using different request-arrival traces. Our results from exhaustive experimental analysis demonstrate that *Cocktail* can minimize deployment cost by 1.4× while meeting the accuracy for up-to 96% of the requests and providing 2× reduction in latency, when compared to state-of-the-art model serving systems.
5. We show that ensemble models are inherently fault-tolerant over single models, since in the former, failure of a model would incur some accuracy loss without complete failure of the requests. It is observed from our failure-resilience results that *Cocktail* can adapt to instance failures by limiting the accuracy loss within 0.6%.

2 Background and Motivation

We start by providing a brief overview of model-serving in public cloud and ensembling, followed by a detailed analysis of their performance to motivate the need for *Cocktail*.

2.1 Model Serving in Public Cloud

Figure 2 shows the overall architecture of a model-serving framework. There are diverse applications that are typically developed, trained and hosted as web services. These services allow end-users to submit queries via web server interface. Since these inference requests are often user-facing, it is imperative to administer them under a strict service level objective (SLO). We define SLO as the end-to-end response latency required by an application. Services like Ads and News Feed [39, 44] would require SLOs within 100ms, while facial tag recommendation [83] can tolerate up to 1000ms. A myriad of model architectures are available to train these applications which by themselves can be deployed on application frameworks like `TensorFlow` [1], `PyTorch` [62] etc. Table 1 shows the different models available for image prediction, that are pretrained on Keras using `ImageNet` [29] dataset. Each model has unique accuracy and latencies depending on the model architecture. Typically denser models are designed with more parameters (ex. `NASLarge`) to classify complex

Model (Acronym)	Params (10k)	Top-1 Accuracy(%)	Latency (ms)	P_f
MobileNetV1 (MNet)	4,253	70.40	43.45	10
MobileNetV2 (MNetV2)	4,253	71.30	41.5	10
NASNetMobile (NASMob)	5,326	74.40	78.18	3
DenseNet121 (DNet121)	8,062	75.00	102.35	3
DenseNet201 (DNet201)	20,242	77.30	152.21	2
Xception (Xcep)	22,910	79.00	119.2	4
Inception V3 (Incep)	23,851	77.90	89	5
ResNet50-V2 (RNet50)	25,613	76.00	89.5	6
Resnet50 (RNet50)	25,636	74.90	98.22	5
IncepResnetV2 (IRV2)	55,873	80.30	151.96	1
NasNetLarge (NasLarge)	343,000	82.00	311	1

Table 1: Collection of pretrained models used for image classification.

Features	Clipper [27]	Rafiki [80]	Infaas [83]	MArk [86]	Sagemaker	Swayam [34]	*Cocktail*
Predictive Scaling	✗	✗	✗	✓	✗	✓	✓
SLO Guarantees	✓	✗	✓	✓	✗	✓	✓
Cost Effective	✗	✗	✓	✓	✗	✗	✓
Ensembling	✓	✓	✗	✗	✓	✗	✓
Heterogeneous Instances	✗	✓	✓	✓	✓	✗	✓
Dynamic ensemble selection	✗	✗	✗	✗	✗	✗	✓
Model abstraction	✓	✓	✓	✗	✗	✗	✓

Table 2: Comparing *Cocktail* with other related frameworks.

classes of images. These 11 models are a representative set to classify all images belonging to 1000 classes in `Imagenet`. Depending on the application type, the maximum ensemble size can vary from tens to hundreds of models.

The entire model framework is typically hosted on resources like VMs or containers in public cloud. These resources are available in different types including CPU/GPU instances, burstables and transient instances. Transient instances [69] are similar to traditional VMs but can be revoked at any time by the cloud provider with an interruption notice. The provisioning latency, instance permanence and packing factor of these resources have a direct impact on the latency and cost of hosting model-serving. We explain instance "packing factor" and its relationship with latency in Section 2.3.2. In this paper, we focus on improving the accuracy and latency from the model selection perspective and consider instances types from a cost perspective. A majority of the model serving systems [6, 83, 86] in public cloud support individual model selection from available models. For instance, InFaas [83] can choose variants among a same model to maintain accuracy and latency requirements. However, denser models tend to have up to 6× the size and twice the latency of smaller models to achieve increased accuracy of about 2-3%. Besides using dense models, ensembling [15] techniques have been used to achieve higher accuracy.

Why Ensembling? An Ensemble is defined as a set of classifiers whose individual decisions combined in some way to classify new examples. This has proved to be more accurate than traditional single large models because it inherently reduces incorrect predictions due to variance and bias. The commonly used ensemble method in classification problems is bagging [33] that considers homogeneous weak learners, learns them independently from each other in parallel, and combines them following some kind of deterministic averaging process [18] or majority voting [49] process. For further details on ensemble models, we refer the reader to prior works [14, 57, 58, 61, 64, 77, 78, 88].

2.2 Related Work

Ensembling in practice: Ensembling is supported by commercial cloud providers like Azure ML-studio [11] and AWS Autogluon [31] to boost the accuracy compared to single models. Azure initially starts with 5 models and scales up to 200 using a hill-climb policy [17] to meet the target accuracy. AWS combines about 6-12 models to give the best possible accuracy. Users also have the option to manually mention the ensemble size. Unlike them, *Cocktail's* model selection policy tries to right-size the ensemble for a given latency, while maximizing accuracy.

Model-serving in Cloud: The most relevant prior works to *Cocktail* are InFaas [83] and Clipper [27], which have been extensively discussed and compared to in Section 6. Recently FrugalML [20] was proposed to cost-effectively choose from commercial MLaaS APIs. While striking a few similarities with *Cocktail*, it is practically limited to image-classification applications with very few classes and does not address resource provisioning challenges. Several works [37, 38] like MArk [86] proposed SLO and cost aware resource procurement policies for model-serving. Although our heterogeneous instance procurement policy has some similarities with MArk, it is significantly different because we consider ensemble models. Rafiki [80] considers small model sets and scales up and down the ensemble size by trading off accuracy to match throughput demands. However, *Cocktail's* resource management is more adaptive to changing request loads and does not drop accuracy. Pretzel [52] and Inferline [26] are built on top of Clipper to optimize the prediction pipeline and cost due to load variations, respectively. Many prior works [2, 25, 35, 63, 74, 75] have extensively tried to reduce model latency by reducing overheads due to shared resources and hardware interference. We believe that our proposed policies can be complementary and beneficial to these prior works to reduce the cost and resource footprint of ensembling. There are mainstream commercial systems which automate single model-serving like TF-Serving [60], SageMaker [6], AzureML [10], Deep-Studio [28] etc.

Autoscaling in Public Cloud: There are several research works that optimize the resource provisioning cost in public cloud. These works are broadly categorized into: (i) multiplexing the different instance types (e.g., Spot, On-Demand) [12, 23, 34, 41, 42, 68, 79], (ii) proactive resource provisioning based on prediction policies [34,36,40,41,69,86]. *Cocktail* uses similar load prediction models and auto-scales VMs in a distributed fashion with respect to model ensembling. Swayam [34] is relatively similar to our work as it han-

Baseline(BL)	NASLarge	IRV2	Xception	DNet121	NASMob
#Models	10	8	7	5	2
BL_Latency	311(ms)	152(ms)	120(ms)	100(ms)	98(ms)
E_Latency	152(ms)	120(ms)	103(ms)	89(ms)	44(ms)

Table 3: Comparing latency of Ensembling (E_Latency) with single (baseline) models.

dles container provisioning and load-balancing, specifically catered for single model inferences. *Cocktail's* autoscaling policy strikes parallels with Swayam's distributed autoscaling; however, we further incorporate novel importance sampling techniques to reduce over-provisioning for under-used models. Table 2 provides a comprehensive comparison of *Cocktail* with the most relevant works across key dimensions.

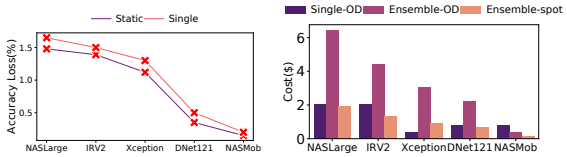
2.3 Pros and Cons of Model Ensembling

In this section, we quantitatively evaluate (i) how effective ensembles are in terms of accuracy and latency compared to single models, and (ii) the challenges in deploying ensemble frameworks in a cost-effective fashion on a public cloud. For relevance in comparison to prior work [27, 83] we chose image inference as our ensemble workload. While ensembling is applicable in other classification workloads like product recommendations [24, 53], text classification [71] etc, the observations drawn are generic and applicable to other applications.

2.3.1 Ensembling Compared to Single Models

To analyze the accuracy offered by ensemble models, we conduct an experiment using 10000 images from `ImageNet` [29] test dataset, on a `C5.xlarge` [8] instances in AWS EC2 [5]. For a given baseline model, we combine all models whose latency is lower than that of the baseline, and call it full-ensemble. We perform ensembling on the predictions using a simple majority voting policy. The latency numbers for the baseline models and the corresponding ensemble models along with the size of the ensemble are shown in Table 3. In majority voting, every model votes for a prediction for each input, and the final output prediction is the one that receives more than half of the votes. Figure 3a, shows the accuracy comparison of the baseline (single) and static ensemble (explained in Section 3) compared to the full-ensemble. It is evident that full-ensemble can achieve up to 1.65% better accuracy than single models.

Besides accuracy again, ensembling can also achieve lower latency. The latency of the ensemble is calculated as the time between start and end of the longest running model.As shown in Table 3, in the case of `NASLarge`, the ensemble latency is 2× lower (151ms) than the baseline latency (311ms). Even a 10ms reduction in latency is of significant importance to the providers [35]. We observe a similar trend of higher ensemble accuracy for other four baseline models with a latency reduction of up to 1.3×. Thus, depending on the model subset used in the ensemble, it achieves better accuracy than the baseline at lower latencies. Note that in our example model-set, the benefits of ensembling will diminish for lower



(a) Accuracy loss compared to full-ensemble.

(b) Cost of full-ensembling hosted on OD and Spot instances.

Figure 3: Cost and accuracy of ensembling vs single models.

accuracies (< 75%) because single models can reach those accuracies. Hence, based on the user constraints, *Cocktail* chooses between ensemble and single models.

2.3.2 Ensembling Overhead

While ensembling can boost accuracy with low latency, their distinctive resource hungry nature drastically increases the deployment costs when compared to single models. This is because more VMs or containers have to be procured to match the resource demands. However, note that the "Packing factor" (P_f) for each model also impacts the deployment costs. P_f in this context is defined as the number of inferences that can be executed concurrently in a single instance without violating the inference latency (on average). Table 1 provides the P_f for 11 different models when executed on a `C5.xlarge` instance. There is a linear relationship between P_f and the instance size. It can be seen that smaller models (`MNet, NASMob`) can be packed 2-5× more when compared to larger models (`IRV2, NASLarge`). Thus, the ensembles with models of higher P_f have significantly lower cost.

The benefits of P_f is contingent upon the models chosen by the model selection policy. Existing ensemble model selection policies used in systems like Clipper use all off-the-shelf models and assign weights to them to calculate accuracy. However, they do not right-size the model selection to include models which primarily contribute to the majority voting. We compare the cost of hosting ensembles using both spot (ensemble-spot) and OD (ensemble-OD) instances with the single models hosted on OD (single-OD) instances. Ensemble-spot is explained further in the next section. We run the experiment over a period of 1 hour for 10 requests/second. The cost is calculated as the cost per hour of EC2 c5.xlarge instance use, billed by AWS [5]. We ensure all instances are fully utilized by packing multiple requests in accordance to the P_f. As shown in Figure 3b, Ensemble-OD is always expensive than single-OD for the all the models. Therefore, it is important to ensemble an "optimal" number of less compute intensive models to reduce the cost.

3 Prelude to Cocktail

To specifically address the cost of hosting an ensembling-based model-serving framework in public clouds without sacrificing the accuracy, this section introduces an overview of the two primary design choices employed in *Cocktail*.

How to reduce resource footprint? The first step towards making model ensembling cost effective is to minimize the

number of models by pruning the ensemble, which reduces the overall resource footprint. In order to estimate the right number of models to participate in a given ensemble, we conduct an experiment where we chose top $\frac{N}{2}$ accurate models (static) from the full-ensemble of size N. From Figure 3a, it can be seen that the static policy has an accuracy loss of up to 1.45% when compared to full-ensemble, but is still better than single models. This implies that the models other than top $\frac{N}{2}$ yields a significant 1.45% accuracy improvement in the full-ensemble but they cannot be statically determined. Therefore, a full-ensemble model participation is not required for all the inputs because, every model is individually suited to classify certain classes of images when compared to other classes. Figure 4 shows the class-wise accuracy for three models on 5 distinct classes. It can be seen that for simpler classes like Slug, `MNetV2` can achieve similar accuracy as the bigger models, while for difficult classes, like Cup and Quill, it experiences up to 3% loss in accuracy. Since the model participation for ensembling can vary based on the class of input images being classified, there is a scope to develop a dynamic model selection policy that can leverage this class-wise variability to intelligently determine the number of models required for a given input.

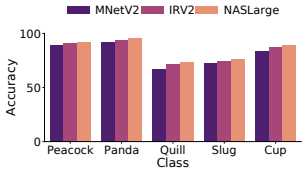


Figure 4: Class-wise Accuracy.

Key Takeaway: *Full ensemble model-selection is an overkill, while static-ensemble leads to accuracy loss. This calls for a dynamic model selection policy which can accurately determine the number of models required, contingent upon the accuracy and scalability of the model selection policy.*

How to save cost? Although dynamic model selection policies can significantly reduce the resource footprint as shown in Figure 3b, the cost is still 20-30% higher when compared to a single model inference. Most cloud providers offer transient VMs such as Amazon Spot instances [69], Google Preemptible VMs [9], and Azure Low-priority VMs [7], that can reduce cloud computing costs by as much as 10× [3]. In *Cocktail*, we leverage these transient VMs such as spot instances to drastically reduce the cost of deploying ensembling model framework. As an example, we host full-ensembling on AWS spot instances. Figure 3b shows that ensemble-spot can reduce the cost by up to 3.3× when compared to ensemble-OD. For certain baselines like IRV2, ensemble-spot is also 1.5× cheaper than single-OD. However, the crucial downside of using transient VMs is that they can be unilaterally preempted by the cloud provider at any given point due to reasons like increase in bid-price or provider-induced random interruptions. As we will discuss further, *Cocktail* is resilient to instance failures owing to the fault-tolerance of ensembling by computing multiple inferences for a single request.

Key takeaway: *The cost-effectiveness of transient instances, is naturally suitable for hosting ensemble models.*

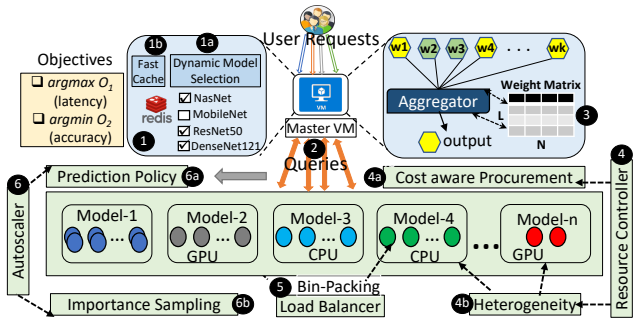


Figure 5: High-level overview of *Cocktail* design.

4 Overall Design of Cocktail

Motivated by our observations, we design a novel model-serving framework, *Cocktail*, that can deliver high-accuracy and low-latency predictions at reduced cost. Figure 5 depicts the high-level design of *Cocktail*. Users submit requests to a master VM, which runs a model selection algorithm, ❶a to decide the models to participate in the ensemble. The participating models are made available in a model cache ❶b for faster access and avoid re-computation for requests having similar constraints. Then, individual queries are dispatched to instances pools ❷ dedicated for each model. The results from the workers are *ensembled* using an weighted majority voting aggregator ❸ to agree upon a correct prediction. To efficiently address the resource management and scalability challenges, *Cocktail* applies multiple strategies.

First, it maintains dedicated instance pools to serve individual models which simplifies the management and load balancing overheads for every model. Next, the resource controller ❹ handles instance procurement, by exploiting both CPU and GPU instances ❹a in a cost-aware ❹b fashion, while the load balancer ❺ ensures all procured instances are bin-packed by assigning queries to appropriate instances. We also design an autoscaler ❻, which utilizes a prediction policy ❻a to forecast the request load and scale instances for every model pool, thereby minimizing over-provisioning of resources. The autoscaler further employs an importance sampling ❻b algorithm to estimate the importance of each model pool by calculating percentage of request served by it in a given time interval. The key components of the design are explained in detail below.

4.1 Dynamic Model Selection Policy

We use a window-based dynamic model selection policy using two objective functions as described below.

Objective functions: In order to reduce cost and latency while maximizing the accuracy, we define a latency-accuracy metric (μ_{AL}) and cost metric (μ_c):

$$\mu_{AL} = \frac{Acc_{target}}{Lat_{target}} \qquad \mu_C = k \times \sum_{m=1}^{N} \frac{inst\_cost}{P_{f_m}}$$

where N is the number of models used to ensemble and $inst\_cost$ is the VM cost. Each model m has a packing factor

P_{f_m} and k is a constant which depends on the VM size in terms of vCPUs (xlarge, 2xlarge, etc). Our first objective function (O_1) is to the maximize μ_{AL} such that target accuracy (Acc_{target}) is reached within the target latency (Lat_{target}).

$$\max \mu_{AL} : \begin{cases} Acc_{target} \geq Acc_{target} \pm Acc_{margin} \\ Lat_{target} \leq Lat_{target} \pm Lat_{margin} \end{cases}$$

To solve O_1, we determine an initial model list by choosing the individual models satisfying Lat_{target} and then create a probabilistic ensemble that satisfies the Acc_{target}. *Cocktail* takes the accuracy of each model as a probability of correctness and then iteratively constructs a model list, where the joint probability of them performing the classification is within the accuracy target. We tolerate a 0.2% (Acc_{margin}) and 5ms (Lat_{margin}) variance in Acc_{target} and Lat_{target}, respectively. Next, we solve for the second objective function (O_2) by minimizing μ_C, while maintaining the target accuracy.

$$\min \mu_C : \{ \quad Acc_{target} \geq Acc_{target} \pm Acc_{margin}$$

(O_2) is solved by resizing the model list of size N and further through intelligence resource procurement (described in section 4.2), and thus maximizing P_f and minimizing k simultaneously. For N models, where each model has a minimum accuracy 'a', we model the ensemble as a coin-toss problem, where N biased coins (with probability of head being a) are tossed together, and we need to find the probability of majority of them being heads. For this, we need at least $\lfloor \frac{N}{2} \rfloor + 1$ models to give the same results. The probability of correct prediction is given by

$$\sum_{i=\lfloor \frac{N}{2} \rfloor + 1}^{N} \binom{N}{i} a^i \ (1-a)^{(N-i)}$$

Model Selection Algorithm: To minimize μ_C, we design a policy to downscale the number of models, if more than N/2+1 models vote for the same classification result. Algorithm 1 describes the overall design of the model selection policy ❶a. For every monitoring interval, we keep track of the accuracy obtained from predicting all input images within the interval. If the accuracy of the interval reaches the threshold accuracy (target + error_margin), we scale down the number of available models in the ensemble. For consecutive sampling intervals, we calculate the `Mode` (most frequently occurring) of the majority vote received for every input. If the `Mode` is greater than needed votes $\lfloor N/2 \rfloor + 1$ we prune the models to $\lfloor N/2 \rfloor + 1$. While down-scaling, we drop the models with the least prediction accuracy in that interval. If there is a tie, we drop the model with least packing factor (P_f). It can so happen that dropping models can lead to drop in accuracy for certain intervals, because the class of images being predicted are different. In such cases, we up-size the models (one at a time) by adding most accurate model from the remaining unused models.

Algorithm 1 Model Selection and Weighted Majority Voting

```
1: procedure FULL_ENSEMBLE(MODELLIST, SLO)
2:     for model ∈ ModelList do
3:         if model.latency ≤ SLO.latency then
4:             Model.add(model)
5:         end if
6:     end for  (O1)
7: end procedure
8: procedure DYNAMIC_MODEL_SCALING(Models)
9:     if curr_accuracy ≥ accuracy_threshold then
10:        if max_vote > N/2 + 1 then (O2)
11:            to_be_dropped ← max_vote − N/2 + 1
12:            Models.drop(to_be_dropped)
13:        end if
14:    else
15:        addModel ← find_models(remaining_models)
16:        Models.append(addModel)
17:    end if
18: end procedure
19: procedure WEIGHTED_VOTING(Models)
20:    for model in ∀Models do
21:        class ← model.predicted_class
22:        weighted_vote[class] += weights[model.class]
23:    end for
24:    P_class ← max(weighted_vote, key = class)
25:    return P_class
26: end procedure
```

4.1.1 Class-based Weighted Majority Voting

The model selection policy described above ensures that we only use the necessary models in the majority voting. In order to increase the accuracy of majority voting, we design a weighted majority voting policy ❸. The weight matrix is designed by considering the accuracy of each model for each class, giving us a weight matrix of $L \times N$ dimension, where L is the number of unique labels and N is the number of models used in the ensemble. The majority vote is calculated as a sum of model-weights for each unique class in the individual prediction of the ensemble. For instance, if there are 3 unique classes predicted by all the ensemble models, we sum the weights for all models of the same class. The class with the maximum weight (P_{class}) is the output of the majority vote. Hence, classes that did not get the highest votes can still be the final output if the models associated with that class has a higher weight, than the combined weights of highest voted class. Unlike commonly used voting policies which assign weights based on overall correct predictions, our policy incorporates class-wise information to the weights, thus making it more adaptable to different images classes.

In order to determine the weight of every class, we use a per-class dictionary that keeps track of the correct predictions of every model per class. We populate the dictionary at runtime to avoid any inherent bias that could result from varying images over time. Similarly, our model selection policy is also changed at runtime based on correct predictions seen during every interval. An important concern in majority voting is tie-breaking. Ties occur when two sets of equal number of models predict a different result. The effectiveness

Algorithm 2 Predictive Weighted Instance Auto Scaling

```
1: procedure WEIGHTED_AUTOSCALING(Stages)
2:     Predicted_load ← DeepARN_Predict(load)
3:     for every Interval do
4:         for model in ∀Models do
5:             model_weight ← get_popularity(model)
6:             Weight.append(model_weight)
7:         end for
8:     end for
9:     if Predicted_load ≥ Current_load then
10:        for model in ∀Models do
11:            I_n ← (Predicted_load - Current_load) × model_weight
12:            launch_workers(est_VMs)
13:            model.workers.append(est_VMs)
14:        end for
15:    end if
16: end procedure
```

of weighted voting in breaking ties is discussed in Section 6.

4.2 Resource Management

Besides model selection, it is crucial to design an optimized resource provisioning and management scheme to host the models cost-effectively. We explain in detail the resource procurement and autoscaling policy employed in *Cocktail*.

4.2.1 Resource Controller

Resource controller determines the cost-effective combination of instances to be procured. We explain the details below.

Resource Types: We use both CPU and GPU instances 4a depending on the request arrival load. GPU instances are cost-effective when packed with a large batch of requests for execution. Hence, inspired from prior work [27, 86], we design an adaptive packing policy such that it takes into account the number of requests to schedule at time T and P_f for every instance. The requests are sent to GPU instances only if the load matches the P_f of the instance.

Cost-aware Procurement: The cost of executing in a fully packed instance determines how expensive is each instance. Prior to scaling-up instances, we need to estimate the cost 4b of running them along with existing instances. At any given time T, based on the predicted load (L_p) and running instances R_N, we use a cost-aware greedy policy to determine the number of additional instances required to serve as $A_n = L_p - C_r$, where $C_r = \sum_{i=1}^{N} P_{f_i}$, is the request load which can be handled with R_N. To procure A_n instances, we greedily calculate the least cost instance as $\min_{\forall i \in instances} Cost_i \times A_n / P_{f_i}$. Depending on the cost-effectiveness ratio of A_n/P_{f_i}, GPUs will be preferred over CPU instances.

Load Balancer: Apart from procuring instances, it is quintessential to design a load balancing and bin-packing 5 strategy to fully utilize all the provisioned instances. We maintain a request queue at every model pool. In order to increase the utilization of all instances in a pool at any given time, the load balancer submits every request from the queue to the lease remaining free slots (viz. instance packing factor P_f). This is similar to an online bin-packing algorithm. We use an idle-timeout limit for 10 minutes to recycle unused instances from every model pool. Hence, greedily assigning requests enables early scale down of lightly loaded instances.

4.2.2 Autoscaler

Along with resource procurement, we need to autoscale instances to satisfy the incoming query load. Though reactive policies (used in Clipper and InFaas) can be employed which take into account metrics like CPU utilization [83], these policies are slow to react when there is dynamism in request rates. Proactive policies with request prediction are know to have superior performance [86] and can co-exist with reactive policies. In *Cocktail*, we use a load prediction model that can accurately forecast the anticipated load for a given time interval. Using the predicted load 6a, *Cocktail* spawns additional instances, if necessary, for every instance pool. In addition, we sample SLO violations for every 10s interval and reactively spawn additional instances to every pool based on aggregate resource utilization of all instances. This captures SLO violations due to mis-predictions.

Prediction Policy: To effectively capture the different load arrival patterns, we design a DeepAR-estimator (DeepARest) based prediction model. We zeroed in on the choice of using DeepARest by conducting (Table 4) an in-depth comparison of the accuracy loss when compared with other state-of-the-art traditional and ML-based prediction models used in prior works [47, 86]. As shown in Algorithm 2, for every model under a periodic scheduling interval of 1 minute (T_s), we use the *Predicted_*load (L_p) at time $T + T_p$ and compare it with the *current_load* to determine the number of instances (I_n).T_p is defined as the average launch time for new instances. (T_s) is set to 1 minute as it is the typical instance provisioning time for EC2 VMs. To calculate (L_p), we sample the arrival rate in adjacent windows of size W over the past S seconds. Using the global arrival rate from all windows, the model predicts (L_p) for T_p time units from T. T_p is set to 10 minutes because it is sufficient time to capture the variations in long-term future. All these parameters are tunable based on the system needs.

Model	RMSE
MWA	77.5
EWMA	88.25
Linear R.	87.5
Logsitic R.	78.34
Simple FF.	45.45
LSTM	28.56
DeepArEst	26.67

Table 4: Prediction models.

Importance Sampling: An important concern in autoscaling is that the model selection policy dynamically determines the models in the ensemble for a given request constraints. Autoscaling the instances equally for every model based on predicted load, would inherently lead to over-provisioned instances for under-used models. To address this concern, we design a weighted autoscaling policy which intelligently auto-scales instances for every pool based on the weights. As shown in Algorithm 2, weights are determined by frequency in which a particular model is chosen for requests (*get_popularity*) with respect to other models in the ensemble.

The weights are multiplied with the predicted load to scale instances *(launch_workers)* for every model pool. We name this as an importance sampling (6b) technique, because the model pools are scaled proportional to their popularity.

5 Implementation and Evaluation

We implemented a prototype of *Cocktail* and deployed it on AWS EC2 [5] platform The details of the implementation are described below. Cocktail is open-sourced at https://github.com/jashwantraj92/cocktail

5.1 Cocktail Prototype Implementation

Cocktail is implemented using 10KLOC of `Python`. We designed *Cocktail* as a client-server architecture, where one master VM receives all the incoming requests which are sent to individual model worker VMs.

Master-Worker Architecture: The master node handles the major tasks such as (i) concord model selection policy, (ii) request dispatch to workers VMs as asynchronous future tasks using `Python asyncio` library, and (iii) ensembling the prediction from the worker VMs. Also all VM specific metrics such as current_load, CPU utilization, etc. reside in the master node. It runs on a `C5.16x` [8] large instance to handle these large volume of diverse tasks. Each worker VMs runs a client process to serve its corresponding model. The requests are served as independent parallel threads to ensure timely predictions. We use `Python Sanic` web-server for communication with the master and worker VMs. Each worker VM runs tensorflow-serving [60] to serve the inference requests.

Load Balancer: The master VMs runs a separate thread to monitor the importance sampling of all individual model pools. It keeps track of the number of requests served per model in the past 5 minutes. This information is used for calculating the weights per model for autoscaling decisions. We integrate a `mongodb` [21] database in the master node to maintain all information about procured instances, spot-instance price list, and instance utilization. The load prediction model resides in the master VM which constantly records the arrival rate in adjacent windows. Recall that the details of the prediction were described in Section 4.2.2. The DeepAREst [4] model was trained using `Keras` [22] and `Tensorflow`, over 100 epochs with 2 layers, 32 neurons and a batch-size of 1.

Model Cache: We keep track of the model selected for ensembling on a per request constraint basis. The constraints are defined as `<latency,accuracy>` pair. The queries arriving with similar constraints can read the model cache to avoid re-computation for selecting the models. The model cache is implemented as a hash-map using `Redis` [16] in-memory key-value store for fast access.

Constraint specification: We expose a simple API to developers, where they can specify the type of inference task (e.g., classification) along with the `<latency,accuracy>` constraints. Developers also need to indicate the primary objective between these two constraints. *Cocktail* automatically chooses a set of single or ensemble models required to meet the developer specified constraints.

Dataset	Application	Classes	Train-set	Test-set
ImageNet [29]	Image	1000	1.2M	50K
CIFAR-100 [50]	Image	100	50K	10K
SST-2 [72]	Text	2	9.6K	1.8K
SemEval [66]	Text	3	50.3K	12.2K

Table 5: Benchmark Applications and datasets.

Discussion: Our accuracy and latency constraints are limited to the measurements from the available pretrained models. Note that changing the models or/and framework would lead to minor deviations. While providing latency and top-1% accuracy of the pretrained models is an offline step in *Cocktail*, we can calculate these values through one-time profiling and use them in the framework. All decisions related to VM autoscaling, bin-packing and load-prediction are reliant on the centralized mongodb database, which can become a potential bottleneck in terms of scalability and consistency. This can be mitigated by using fast distributed solutions like Redis [16] and Zookeeper [46]. The DeepARest model is pre-trained using 60% of the arrival trace. For varying load patterns, the model parameters can be updated by re-training in the background with new arrival rates.

5.2 Evaluation Methodology

We evaluate our prototype implementation on AWS EC2 [8] platforms. Specifically, we use `C5.xlarge, 2xlarge, 4xlarge, 8xlarge` for CPU instances and `p2.xlarge` for GPU instances.

Load Generator: We use different traces which are given as input to the load generator. Firstly, we use real-world request arrival traces from Wikipedia [76], which exhibit typical characteristics of ML inference workloads as it has recurring diurnal patterns. The second trace is production twitter [48] trace which is bursty with unexpected load spikes. We use the first 1 hour sample of both the traces and they are scaled to have an average request rate of 50 req/sec.

Workload: As shown in Table 5 we use image-classification and Sentiment Analysis (text) applications with two datasets each for our evaluation. Sentiment analysis outputs the sentiment of a given sentence as positive negative and (or) neutral. We use 9 different prominently used text-classification models from transformers library [81] (details available in appendix) designed using Google BERT [30] architecture trained on `SST` [72] and `SemEval` [66] dataset. Each request from the load-generator is modelled after a query with specific `<latency,accuracy>` constraints. The queries consist of images or sentences, which are randomly picked from the test dataset. In our experiments, we use five different types of these constraints.

As an example for the `Imagenet` dataset shown in Figure 6, each constraint is a representative of <latency, accuracy> combination offered by single models (shown in Table 1). We use one constraint (blue dots) each from five different regions

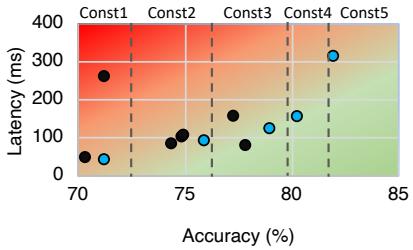


Figure 6: Constraints used in our workloads.

(categorized by dotted lines) picked in the increasing order of accuracy. Each of these picked constraints (named const1 - const5 in the Figure) represents a single baseline model, whose corresponding ensemble size ranges from small (2) to large (10), as shown in Table 3. Note that the latency is the raw model execution latency, and does not include the additional network-transfer overheads incurred. We picked the constraints using a similar procedure by ordering constraints across five different categories for `CIFAR-100`, `SST-2` and `SemEval` (twitter tweets) datasets. The list of models used for them are given in the Appendix. We model two different workload mixes by using a combination of these five query constraint types. Based on the decreasing order of accuracy, we categorize them into *Strict* and *Relaxed* workloads.

5.2.1 Evaluation Metrics

Most of our evaluations of *Cocktail* for image-classification are performed using the `Imagenet` dataset. To further demonstrate the sensitivity of Cocktail to dataset and applicability to other classification applications, we also evaluate it using `CIFAR-100` and Sentiment-Analysis application. We use three important metrics: response latency, cost and accuracy for evaluating and comparing our design to other state-of-the-art systems. The response latency metric includes model inference latency, communication/network latency and synchronization overheads. Queries that do not meet response latency requirements (>700ms) are considered as SLO violations. The cost metric is the billing cost from AWS, and the accuracy metric is measured as the percentage of requests that meet the target accuracy requirements.

We compare these metrics for *Cocktail* against (i) *InFaas* [83], which is our baseline that employs single model selection policy; (ii) *Clipper* [27], which uses static full model selection policy (analogous to AWS AutoGluon); and (iii) *Clipper-X* which is an enhancement to *Clipper* with a simple model selection (drop one model at a time) that does not utilize the *mode*-based policy enforced in *Cocktail*. Both *InFaas* and *Clipper* share *Cocktail*'s implementation setup to ensure a fair comparison with respect to our design and execution environment. For instance, both *Clipper* and *InFaas* employ variants of a reactive autoscaler as described in Section 4.2.2. However, in our setup, both benefit from the distributed autoscaling and prediction policies, thus eliminating variability. Also note that *InFaas* is deployed using OnDemand instances, while both *Clipper* and *Cocktail* use spot instances.

6 Analysis of Results

This section discusses the experimental results of *Cocktail* using the Wiki and Twitter traces. To summarize the overall results, Cocktail providing 2× reduction in latency, while meeting the accuracy for up-to 96% of the requests under reduced deployment cost by 1.4×, when compared to *InFaaS* and *Clipper*.

6.1 Latency, Accuracy and Cost Reduction

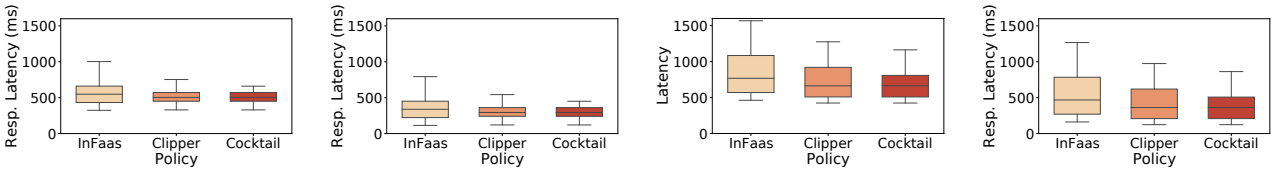
Latency Distribution: Figure 7 shows the distribution of total response latency in a standard box-and-whisker plot. The boundaries of the box-plots depict the 1st quartile (25th percentile (PCTL)) and 3rd quartile (75th PCTL), the whiskers plot the minimum and maximum (tail) latency and the middle line inside the box depict the median (50 PCTL). The total response latency includes additional 200-300ms incurred for query serialization and data transfer over network. It can be seen that the maximum latency of *Cocktail* is similar to the 75th PCTL latency of *InFaas*. This is because the single model inference have up to 2x higher latency to achieve higher accuracy. Consequently, this leads to 35% SLO violations for *InFaas* in the case of Strict workload. In contrast, both *Cocktail* and *Clipper* can reach the accuracy at lower latency due to ensembling, thus minimizing SLO violations to 1%.

Also, the tail latency is higher for Twitter trace (Figure 7c, 7d) owing to its bursty nature. Note that the tail latency of *Clipper* is still higher than *Cocktail* because *Clipper* ensembles more models than *Cocktail*, thereby resulting in straggler tasks in the VMs. The difference in latency between *Cocktail* and *InFaas* is lower for *Relaxed* workload when compared to *Strict* workload (20% lower in tail). Since the *Relaxed* workload has much lower accuracy constraints, smaller models are able to singularly achieve the accuracy requirements at lower latency.

Accuracy violations: The accuracy is measured as a moving window average with size 200 for all the requests in the workload. Both *Clipper* and *Cocktail* can meet the accuracy for 56% of requests, which is 26% and 9% more than *InFaas* and *Clipper* respectively. This is because, intuitively ensembling leads to higher accuracy than single models. However, *Cocktail* is still 9% better than *Clipper* because the class-based weighted voting, is efficient in breaking ties when compared to weighting averaging used in *Clipper*. Since majority voting can include ties in votes, we analyzed the number of ties, which were correctly predicted for all the queries. *Cocktail* was able to deliver correct predictions for 35% of the tied votes, whereas breaking the ties in *Clipper* led only to 20% correct predictions.

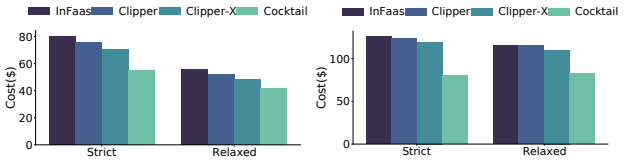
Scheme	Accuracy Met (%)	
	Strict	Relaxed
InFaas	21	71
Clipper	47	89
Cocktail	56	96

Table 6: Requests meeting target accuracy averaged for both Trace.



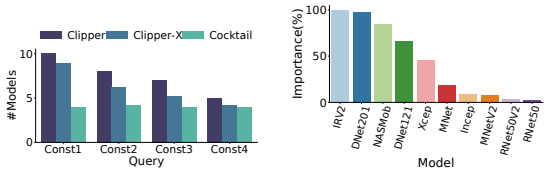
(a) Wiki-trace: *Strict* workload. **(b)** Wiki-trace: *Relaxed* workload. **(c)** Twitter-trace: *Strict* workload. **(d)** Twitter-trace: *Relaxed* workload.

Figure 7: Latency Distribution of *InFaas*, *Clipper* and *Cocktail* for two workload mixes using both Wiki and Twitter traces.



(a) Wiki Trace. **(b)** Twitter Trace.

Figure 8: Cost savings of *Cocktail* compared to three schemes.



(a) Average number of models used in the ensemble. **(b)** Distribution of requests served by each individual model.

Figure 9: Benefits of dynamic model selection policy.

Note that, changing the target accuracy to tolerate a 0.5% loss, increases the percentage of requests that meet accuracy to 81% for *Cocktail*, when compared to 61% for *InFaas*. The requests meeting accuracy are generally higher for the *Relaxed* workload because the target accuracy is much lower. Overall, *Cocktail* was able to deliver an accuracy of 83% and 79.5% on average for the *Strict* and *Relaxed* workloads, respectively. This translates to 1.5% and 1% better accuracy than *Clipper* and *InFaas*. We do not plot the results for *Clipper-X*, which achieves similar accuracy to *Cocktail*, but uses more models as explained in Section 6.2.1.

Cost Comparison: Figure 8 plots the cost savings of *Cocktail* when compared to *InFaas*, *Clipper* and *Clipper-X* policies. It can be seen that, *Cocktail* is up to 1.45× more cost effective than *InFaas* for *Strict* workload. In addition, *Cocktail* reduces cost by 1.35× and 1.27× compared to *Clipper* and *Clipper-X* policies, owing to its dynamic model selection policy, which minimizes the resource footprint of ensembling. On the other hand, *Clipper* uses all models in ensemble and the *Clipper-X* policy does not right size the models as aggressively as *Clipper*, hence they are more expensive. Note that, all the schemes incur higher cost for twitter trace (Figure 8b) compared to wiki trace (Figure 8a). This is because the twitter workload is bursty, thereby leading to intermittent over-provisioned VMs.

6.2 Key Sources of Improvements

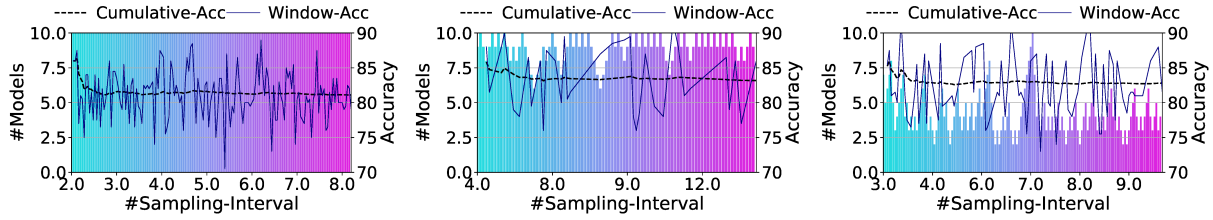
The major improvements in terms of cost, latency, and accuracy in *Cocktail* are explained below. For brevity in explanation, the results are averaged across Wiki and Twitter traces for strict workload.

6.2.1 Benefits from dynamic model selection

Figure 9a plots the average number of models used for queries falling under the first four different constraint (const) types. Here, *Cocktail* reduces the number of models by up to 55% for all four query types. This is because our dynamic policy ensures that the number of models are well within $N/2$ most of the time, whereas the *Clipper-X* policy does not aggressively scale down models. *Clipper*, on the other hand, is static and always uses all the models. The percentage of model-reduction is lower for *Const2*, 3 and 4 because, the total models used in the ensemble is less than *Const1* (8, 7 and 6 models, respectively). Still, the savings in terms of cost will be significant because even removing one model from the ensemble amounts to ∼20% cost savings in the long run (*Clipper* vs *Clipper-X* ensemble in Figure 8). Thus, the benefits of *Cocktail* are substantial for large ensembles while reducing the number of models for medium-sized ensembles.

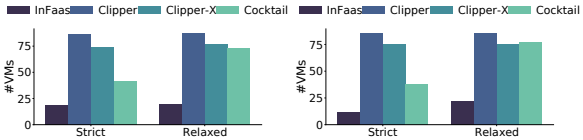
Figure 9b shows the breakdown of the percentage of requests (*Const1*) served by the each model. As seen, InceptionResNetV2, Densenet-201, Densenet121, NasnetMobile and Xception are the top-5 most used models in the ensemble. Based on Table 1, if we had statically taken the top $N/2$ most accurate models, NasNetmobile would not have been included in the ensemble. However, based on the input images sent in each query, our model selection policy has been able to identify NasNetMobile to be a significantly contributing model in the ensemble. Further, the other 5 models are used by up to 25% of the images. Not including them in the ensemble would have led to severe loss in accuracy. But, our dynamic policy with the class-based weighted voting, adapts to input images in a given interval by accurately selecting the best performing model for each class. To further demonstrate the effectiveness of our dynamic model selection,

Figure 10b,10c plots the number models in every sampling interval along with cumulative accuracy and window accuracy within each sampling interval for three schemes. We observe that *Cocktail* can effectively scale up and scale down the models while maintaining the cumulative accuracy well within the threshold. More than 50% of the time the number of models are maintained between 4 to 5, because the dynamic policy is quick in detecting accuracy failures and recovers immediately



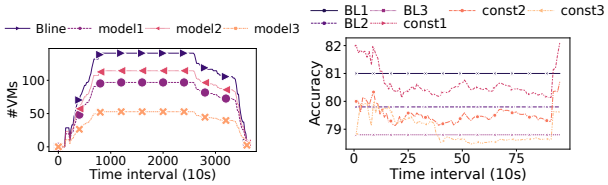
(a) *Clipper* **(b)** *Clipper-X* **(c)** *Cocktail*

Figure 10: Figures (a), (b) and (c) shows the number of models used in ensemble with corresponding cumulative accuracy and window accuracy over a 1 hour period for requests under *Const1*. Figure (d) shows the effects of distributed autoscaling with importance sampling.



(a) Wiki Trace. **(b)** Twitter Trace.

Figure 11: Number of VMs spawned for all four schemes.



(a) Cumulative #VMs. **(b)** Failure Analysis.

Figure 12: Sensitivity analysis of VMs.

by scaling up models. However, *Clipper-X* does not scale down models as frequently as *Cocktail*, while ensuring similar accuracy. *Clipper* is less accurate than *Cocktail* and further it uses all 10 models throughout.

6.2.2 Benefits from Autoscaling

Figure 11 plots the reduction in the number of VMs used by all four schemes. It can be seen that both *Cocktail* and *Clipper-X* spawn 49% and 20% fewer VMs than *Clipper* for workload-1 on Twitter trace. *Cocktail* spawns 29% lesser VMs on top of *Clipper-X*, because it is not aggressive enough like *Cocktail* to downscale more models at every interval. It is to be noted that the savings are lower for *Relaxed* workload because, the number of models in the ensemble are inherently low, thus leading to reduced benefits from scaling down the models. Intuitively, *InFaas* has the least number of VMs spawned because it does not ensemble models. *Cocktail* spawns upto 50% more VMs than *InFaas*, but in turns reduces accuracy loss by up to 96%.

To further capture the benefits of the weighted autoscaling policy, Figure 12a plots the number of VMs spawned over time for the top-3 most used models in the ensemble for *Const1*. The Bline denotes number of VMs that would be spawned without applying the weights. Not adopting an importance sampling based weighted policy would result in equivalent number of VMs as the Bline for all models. However, since *Cocktail* exploits importance sampling by keeping track of the frequency in which models are selected, the number of VMs spawned for model1, model2 and model-3 is upto $3\times$ times lesser than uniform scaling. Figure 9b shows the most used models in decreasing order of importance. The autoscaling policy effectively utilizes this importance factor in regular intervals of 5 minutes. Despite using multiple models for a single inference, importance sampling combined with aggressive model pruning, greatly reduces the resource footprint which directly translates to the cost savings in *Cocktail*.

6.2.3 Benefits of Transient VMs

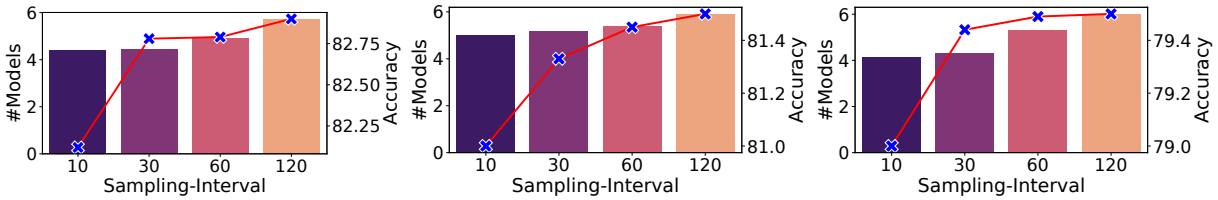
The cost-reductions in *Cocktail* are akin to cost-savings of transient VMs compared to On-Demand (OD) VMs. We profile the spot price of 4 types of `C5` EC2 VMs over a 2-week period in August 2020. It was seen that, the spot instance prices have predictable fluctuations. When compared to the OD price , they were up to 70% cheaper. This price gap is capitalized in *Cocktail* to reduce the cost of instances consumed by ensembling. Note that, we set the bidding price conservatively to 40% of OD. Although, *Cocktail* spawns about 50% more VMs than *InFaas*, the high P_f of small models and spot-instance price reductions combined with autoscaling policies lead to the overall 30-40% cost savings.

6.3 Sensitivity Analysis

In this section, we analyze the sensitivity of *Cocktail* with respect to various design choices which include (i) sampling interval of the accuracy measurements, (ii) spot-instance failure rate and (iii) type of datasets and applications.

6.3.1 Sampling Interval

To study the sensitivity with respect to the sampling interval for measure accuracy loss/gain, we use four different intervals of 10s, 30s, 60s and 120s. Figure 13 plots the average number of models (bar- left y-axis) and cumulative accuracy (line- right y-axis) for the different sampling intervals for queries with three different constraints. It can be seen that the 30s interval strikes the right balance with less than 0.2% loss in accuracy and has average number models much lesser than other intervals. This is because, increasing the interval leads to lower number of scale down operations, thus resulting in a bigger ensemble. As a result, the 120s interval has the highest number of models.



(a) Queries under Constraint-1. **(b)** Queries under Constraint-2. **(c)** Queries under Constraint-3.

Figure 13: Sensitivity analysis of model selection with respect to sampling interval. The average number of models is in primary axis and cumulative accuracy in secondary axis.

6.3.2 Cocktail Failure Resilience

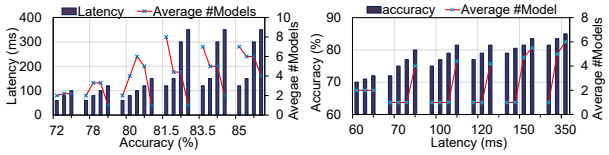
We use spot instances to host models in *Cocktail*. As previously discussed in Section 3, spot instances interruptions can lead to intermittent loss in accuracy as certain models will be unavailable in the ensemble. However for large ensembles (5 models are more), the intermittent accuracy loss is very low. Figure 12b plots the failure analysis results for top three constraints by comparing the ensemble accuracy to the target accuracy. The desired accuracy for all three constraints are plotted as BL1, BL2 and BL3. We induce failures in the instances using *chaosmonkey* [19] tool with a 20% failure probability. It can be seen that queries in all three constraints suffer an intermittent loss in accuracy of 0.6% between the time period 240s and 800s. Beyond 800s, they quickly recover back to the required accuracy because additional instances are spawned in place of failed instances. However, in the case of *InFaas*, this would lead to 1% failed requests due to requests being dropped from the failed instances.

An alternate solution would be to restart the queries in running instances but that leads to increased latencies for the 1% requests. In contrast, *Cocktail* incurs a modest accuracy loss of well within 0.6% and quickly adapts to reach the target accuracy. Thus, *Cocktail* is inherently fault-tolerant owing to the parallel nature in computing multiple inferences for a single request. We observe similar accuracy loss or lower for different probability failures of 5%, 10% and 25%, respectively (results/charts omitted in the interest of space).

Discussion: For applications that are latency tolerant, we can potentially redirect requests from failed instances to existing instances, which would lead to increased tail latency. The results we how are only for latency intolerant applications. Note that, the ensembles used in our experiments are at-least 4 models or more. For smaller ensembles, instance failures might lead to higher accuracy loss, but in our experiments, single models typically satisfy their constraints.

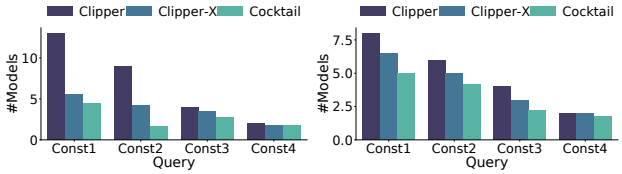
6.3.3 Sensitivity to Constraints

Figure 14 plots the sensitivity of model selection policy under a wide-range of latency and accuracy constraints. In Figure 14a, we vary the latency under six different constant accuracy categories. It can be seen that for fixed accuracy of 72%, 78% and 80%, the average number of models increase with increase in latency, but drops to 1 for the highest latency. Intuitively, singe large models with higher latency can satisfy



(a) Fixed Accuracy. **(b)** Fixed Latency.

Figure 14: Sensitivity Constraints under fixed latency and accuracy. Bar graphs (latency) plotted using primary y-axis and line graph (#models) plotted using secondary y-axis.



(a) Image Classification-Cifar-100. **(b)** Sentiment analysis.

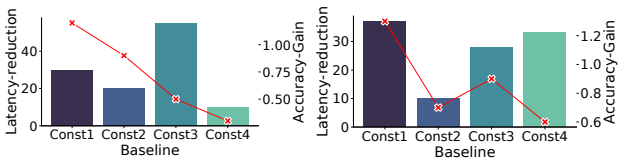
Figure 15: Average number of models used in the ensemble.

the accuracy, while short latency models need to be ensembled to reach the same accuracy. For accuracy greater than 80%, the ensemble size drops with higher latencies. This is because the models which offer higher accuracy are typically dense and hence, smaller ensembles are sufficient. In Figure 14b, we vary the accuracy under six different constant latency categories. It can be seen that for higher accuracies, *Cocktail* tries to ensemble more models to reach the accuracy, while for lower accuracy it resorts to using single models.

6.3.4 Sensitivity to Dataset

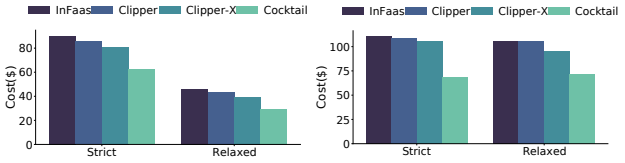
To demonstrate the applicability of *Cocktail* to multiple datasets, we conducted similar experiments as elucidated in Section 5.2.1 using the `CIFAR-100` dataset [50]. It comprises of 100 distinct image classes and we trained 11 different models including the nine that are common from Table 1. Figure 15a plots the average number of models used by the three policies for the top four constraints. It can be seen that *Cocktail* shows similar reduction (as Imagenet) while using only 4.4 models on average. As expected, *Clipper* and *Clipper-X* use more models than *Cocktail* (11 and 5.4, respectively) due to non-aggressive scaling down of the models used.

Figure 16a plots the latency reduction and accuracy boost when compared to *InFaaS* (baseline). While able to reduce 60% of the models used in the ensemble, *Cocktail* also reduces latency by up to 50% and boosts accuracy by up to 1.2%. *Cocktail* was also able to deliver modest accuracy gain



(a) Image Classification:Cifar100. **(b)** Sentiment Analysis.

Figure 16: Latency reduction (%) plotted as bar graph(primary y-axis) and accuracy gains (%) plotted as line graph (secondary y-axis) over InFaaS.



(a) Wiki Trace. **(b)** Twitter Trace.

Figure 17: Cost savings of Cocktail for Sentiment Analysis.

of 0.5% than *Clipper* (not plotted). The accuracy gain seen in `CIFAR-100` is lesser than ImageNet dataset because the class-based weighted voting works effectively when handling large number of classes (100 in `CIFAR` vs 1000 in ImageNet). Nevertheless, *Cocktail* is able to deliver the accuracy at 2x lower latency than *InFaaS* and 1.35x lower cost than Clipper.

6.4 General Applicability of Cocktail

To demonstrate the general applicability of *Cocktail* to other classification tasks, we evaluated *Cocktail* using a Sentiment Analysis application for two datasets. The results reported are averaged across both the datasets. Figure 15b plots the average number of models used by the three policies for the top four constraints. As shown for *Const1*, *Cocktail* shows similar reduction (as image-classification) with only using 4.8 models on average, which is 40% and 26% lower than *Clipper* and *Clipper-X*, respectively. *Cocktail* is also able to reduce the number of models by 30% and 50% for medium ensembles (*Const2* & *Const3*) as well.

Figure 16b plots the latency reduction and accuracy gain, compared to *InFaaS* (baseline). While being able to reduce 50% of the models used in the ensemble, *Cocktail* also reduces latency by up to 50% and improves accuracy by up to 1.3%. Both *Cocktail* and *Clipper* deliver the same overall accuracy (96%, 94.5%, 93.5%, and 92%)). Since sentiment analysis only has 2-3 classes, there are no additional accuracy gains by using the class-based weighted voting. However, the model selection policy effectively switches between different models based on the structure of input text (equivalent to classes in images). For instance, complex sentences are more accurately classified by denser models compared to smaller. Despite the lower accuracy gains, *Cocktail* is able to reduce the cost (Figure 17) of model-serving by 1.45× and 1.37× for Wiki trace compared to *InFaaS* and *Clipper*, respectively.

7 Concluding Remarks

There is an imminent need to develop model serving systems that can deliver highly accurate, low latency predictions at reduced cost. In this paper, we propose and evaluate *Cocktail*, a cost-effective model serving system that exploits ensembling techniques to meet high accuracy under low latency goals. In *Cocktail*, we adopt a three-fold approach to reduce the resource footprint of model ensembling. More specifically, we (i) develop a novel dynamic model selection, (ii) design a prudent resource management scheme that utilizes weighted autoscaling for efficient resource allocation, and (iii) leverage transient VM instances to reduce the deployment costs. Our results from extensive evaluations using both CPU and GPU instances on AWS EC2 cloud platform demonstrate that *Cocktail* can reduce deployment cost by 1.4×, while reducing latency by 2× and satisfying accuracy for 96% of requests, compared to the state-of-the-art model-serving systems.

Acknowledgments

We are indebted to our shepherd Manya Ghobadi, the anonymous reviewers and Anup Sarma for their insightful comments to improve the clarity of the presentation. Special mention to Nachiappan Chidambaram N. for his intellectual contributions. This research was partially supported by NSF grants #1931531, #1955815, #1763681, #1908793, #1526750, #2116962, #2122155, #2028929 ,and we thank NSF Chameleon Cloud project CH-819640 for their generous compute grant. All product names used in this publication are for identification purposes only and may be trademarks of their respective companies.

References

[1] Martín Abadi. Tensorflow: learning functions at scale. In *Acm Sigplan Notices*. ACM, 2016.

[2] Deepak Agarwal, Bo Long, Jonathan Traupman, Doris Xin, and Liang Zhang. Laser: A scalable response prediction platform for online advertising. In *Proceedings of the 7th ACM international conference on Web search and data mining*, pages 173–182, 2014.

[3] Ahmed Ali-Eldin, Jonathan Westin, Bin Wang, Prateek Sharma, and Prashant Shenoy. Spotweb: Running latency-sensitive distributed web services on transient cloud servers. In *Proceedings of the 28th International Symposium on High-Performance Parallel and Distributed Computing*, pages 1–12, 2019.

[4] Amazon. Deepar estimator. `https://docs.aws.amazon.com/sagemaker/latest/dg/deepar.html,February2020`.

[5] Amazon. EC2 pricing. https://aws.amazon.com/ec2/pricing/.

[6] Amazon. Sagemaker. https://aws.amazon.com/sagemaker/, February 2018.

[7] Amazon. Azure Low priority batch VMs., February 2018. https://docs.microsoft.com/en-us/azure/batch/batch-low-pri-vms .

[8] Amazon. EC2 C5 Instances., February 2018. https://aws.amazon.com/ec2/instance-types/c5/ .

[9] Amazon. Google Preemptible VMs., February 2018. https://cloud.google.com/preemptible-vms .

[10] Azure. Machine Learning as a Service., February 2018. https://azure.microsoft.com/en-us/pricing/details/machine-learning-service/ .

[11] Azure. Ensembling in Azure ML Studio., February 2020. https://docs.microsoft.com/en-us/azure/machine-learning/studio-module-reference/multiclass-decision-forest .

[12] Ataollah Fatahi Baarzi, Timothy Zhu, and Bhuvan Urgaonkar. Burscale: Using burstable instances for cost-effective autoscaling in the public cloud. In *Proceedings of the ACM Symposium on Cloud Computing*, New York, NY, USA, 2019. Association for Computing Machinery.

[13] Marian Stewart Bartlett, Gwen Littlewort, Mark Frank, Claudia Lainscsek, Ian Fasel, and Javier Movellan. Recognizing facial expression: machine learning and application to spontaneous behavior. In *2005 IEEE Computer Society Conference on Computer Vision and Pattern Recognition (CVPR'05)*, volume 2, pages 568–573. IEEE, 2005.

[14] Eric Bauer and Ron Kohavi. An empirical comparison of voting classification algorithms: Bagging, boosting, and variants. *Machine learning*, 36(1-2):105–139, 1999.

[15] William H Beluch, Tim Genewein, Andreas Nürnberger, and Jan M Köhler. The power of ensembles for active learning in image classification. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition*, pages 9368–9377, 2018.

[16] Josiah L Carlson. *Redis in action*. Manning Publications Co., 2013.

[17] Rich Caruana, Alexandru Niculescu-Mizil, Geoff Crew, and Alex Ksikes. Ensemble selection from libraries of models. In *Proceedings of the twenty-first international conference on Machine learning*, page 18, 2004.

[18] Jesús Cerquides and Ramon López De Mántaras. Robust bayesian linear classifier ensembles. In *European Conference on Machine Learning*, pages 72–83. Springer, 2005.

[19] Michael Alan Chang, Bredan Tschaen, Theophilus Benson, and Laurent Vanbever. Chaos monkey: Increasing sdn reliability through systematic network destruction. In *Proceedings of the 2015 ACM Conference on Special Interest Group on Data Communication*, pages 371–372, 2015.

[20] Lingjiao Chen, Matei Zaharia, and James Zou. Frugalml: How to use ml prediction apis more accurately and cheaply. In *Advances in Neural Information Processing Systems (NeurIPS)*, 2020.

[21] Kristina Chodorow. *MongoDB: the definitive guide: powerful and scalable data storage*. " O'Reilly Media, Inc.", 2013.

[22] Francois Chollet. *Deep Learning mit Python und Keras: Das Praxis-Handbuch vom Entwickler der Keras-Bibliothek*. MITP-Verlags GmbH & Co. KG, 2018.

[23] Andrew Chung, Jun Woo Park, and Gregory R. Ganger. Stratus: Cost-aware container scheduling in the public cloud. In *SoCC*, 2018.

[24] Paul Covington, Jay Adams, and Emre Sargin. Deep neural networks for youtube recommendations. In *Proceedings of the 10th ACM conference on recommender systems*, pages 191–198, 2016.

[25] Daniel Crankshaw, Peter Bailis, Joseph E Gonzalez, Haoyuan Li, Zhao Zhang, Michael J Franklin, Ali Ghodsi, and Michael I Jordan. The missing piece in complex analytics: Low latency, scalable model management and serving with velox. *arXiv preprint arXiv:1409.3809*, 2014.

[26] Daniel Crankshaw, Gur-Eyal Sela, Corey Zumar, Xiangxi Mo, Joseph E. Gonzalez, Ion Stoica, and Alexey Tumanov. Inferline: ML inference pipeline composition framework. *CoRR*, abs/1812.01776, 2018.

[27] 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, Boston, MA, March 2017. USENIX Association.

[28] Deepstudio. Deep Learning Dtudio, February 2020. https://docs.deepcognition.ai/ .

[29] J. Deng, W. Dong, R. Socher, L. Li, and and. Imagenet: A large-scale hierarchical image database. In *2009 IEEE Conference on Computer Vision and Pattern Recognition*, June 2009.

[30] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding, 2019.

[31] Nick Erickson, Jonas Mueller, Alexander Shirkov, Hang Zhang, Pedro Larroy, Mu Li, and Alexander Smola. Autogluon-tabular: Robust and accurate automl for structured data, 2020.

[32] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

[33] Mikel Galar, Alberto Fernandez, Edurne Barrenechea, Humberto Bustince, and Francisco Herrera. A review on ensembles for the class imbalance problem: Bagging-, boosting-, and hybrid-based approaches. *IEEE Transactions on Systems, Man, and Cybernetics, Part C (Applications and Reviews)*, 42(4):463–484, 2012.

[34] Arpan Gujarati, Sameh Elnikety, Yuxiong He, Kathryn S. McKinley, and Björn B. Brandenburg. Swayam: Distributed Autoscaling to Meet SLAs of Machine Learning Inference Services with Resource Efficiency. In *USENIX Middleware Conference*, 2017.

[35] Arpan Gujarati, Reza Karimi, Safya Alzayat, Antoine Kaufmann, Ymir Vigfusson, and Jonathan Mace. Serving dnns like clockwork: Performance predictability from the bottom up. In *14th USENIX Symposium on Operating Systems Design and Implementation (OSDI 20)*, Banff, Alberta, November 2020. USENIX Association.

[36] Jashwant Raj Gunasekaran, Prashanth Thinakaran, Nachiappan C.Nachiappan, Mahmut Taylan Kandemir, and Chita R. Das. Fifer: Tackling Resource Underutilization in the Serverless Era. In *USENIX Middleware Conference*, 2020.

[37] Jashwant Raj Gunasekaran, Prashanth Thinakaran, Mahmut Taylan Kandemir, Bhuvan Urgaonkar, George Kesidis, and Chita Das. Spock: Exploiting serverless functions for slo and cost aware resource procurement in public cloud. In *IEEE CLOUD*, 2019.

[38] Jashwant Raj Gunasekaran, Prashanth Thinakaran, Cyan Subhra Mishra, Mahmut Taylan Kandemir, and Chita R. Das. Towards designing a self-managed machine learning inference serving system inpublic cloud, 2020.

[39] U. Gupta, S. Hsia, V. Saraph, X. Wang, B. Reagen, G. Wei, H. S. Lee, D. Brooks, and C. Wu. Deeprecsys: A system for optimizing end-to-end at-scale neural recommendation inference. In *2020 ACM/IEEE 47th Annual International Symposium on Computer Architecture (ISCA)*, pages 982–995, 2020.

[40] Rui Han, Moustafa M. Ghanem, Li Guo, Yike Guo, and Michelle Osmond. Enabling cost-aware and adaptive elasticity of multi-tier cloud applications. *Future Gener. Comput. Syst.*, 32(C):82–98, March 2014.

[41] Aaron Harlap, Andrew Chung, Alexey Tumanov, Gregory R. Ganger, and Phillip B. Gibbons. Tributary: spot-dancing for elastic services with latency SLOs. In *ATC*, 2018.

[42] Aaron Harlap, Alexey Tumanov, Andrew Chung, Gregory R. Ganger, and Phillip B. Gibbons. Proteus: Agile ML Elasticity Through Tiered Reliability in Dynamic Resource Markets. In *Eurosys*, 2017.

[43] Johann Hauswald, Michael A. Laurenzano, Yunqi Zhang, Cheng Li, Austin Rovinski, Arjun Khurana, Ronald G. Dreslinski, Trevor Mudge, Vinicius Petrucci, Lingjia Tang, and Jason Mars. Sirius: An open end-to-end voice and vision personal assistant and its implications for future warehouse scale computers. In *ASPLOS*, 2015.

[44] K. Hazelwood, S. Bird, D. Brooks, S. Chintala, U. Diril, D. Dzhulgakov, M. Fawzy, B. Jia, Y. Jia, A. Kalro, J. Law, K. Lee, J. Lu, P. Noordhuis, M. Smelyanskiy, L. Xiong, and X. Wang. Applied machine learning at facebook: A datacenter infrastructure perspective. In *2018 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 620–629, Feb 2018.

[45] Andrew Howard, Mark Sandler, Grace Chu, Liang-Chieh Chen, Bo Chen, Mingxing Tan, Weijun Wang, Yukun Zhu, Ruoming Pang, Vijay Vasudevan, et al. Searching for mobilenetv3. In *Proceedings of the IEEE International Conference on Computer Vision*, pages 1314–1324, 2019.

[46] Patrick Hunt, Mahadev Konar, Flavio Paiva Junqueira, and Benjamin Reed. Zookeeper: Wait-free coordination for internet-scale systems. In *USENIX annual technical conference*, 2010.

[47] Minoru Kawashima, Charles E Dorgan, and John W Mitchell. Hourly thermal load prediction for the next 24 hours by arima, ewma, lr and

an artificial neural network. Technical report, American Society of Heating, Refrigerating and Air-Conditioning Engineers . . . , 1995.

[48] Abeer Abdel Khaleq and Ilkyeun Ra. Cloud-based disaster management as a service: A microservice approach for hurricane twitter data analysis. In *GHTC*, 2018.

[49] J Zico Kolter and Marcus A Maloof. Dynamic weighted majority: An ensemble method for drifting concepts. *Journal of Machine Learning Research*, 8(Dec):2755–2790, 2007.

[50] Alex Krizhevsky, Vinod Nair, and Geoffrey Hinton. Cifar-100 (canadian institute for advanced research), 2010. http://www.cs.toronto.edu/~kriz/cifar.html.

[51] Zhenzhong Lan, Mingda Chen, Sebastian Goodman, Kevin Gimpel, Piyush Sharma, and Radu Soricut. Albert: A lite bert for self-supervised learning of language representations. *arXiv preprint arXiv:1909.11942*, 2019.

[52] Yunseong Lee, Alberto Scolari, Byung-Gon Chun, Marco Domenico Santambrogio, Markus Weimer, and Matteo Interlandi. PRETZEL: Opening the black box of machine learning prediction serving systems. In *13th USENIX Symposium on Operating Systems Design and Implementation (OSDI 18)*, pages 611–626, Carlsbad, CA, October 2018. USENIX Association.

[53] Romain Lerallut, Diane Gasselin, and Nicolas Le Roux. Large-scale real-time product recommendation at criteo. In *Proceedings of the 9th ACM Conference on Recommender Systems*, pages 232–232, 2015.

[54] Weibo Liu, Zidong Wang, Xiaohui Liu, Nianyin Zeng, Yurong Liu, and Fuad E Alsaadi. A survey of deep neural network architectures and their applications. *Neurocomputing*, 234:11–26, 2017.

[55] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. Roberta: A robustly optimized bert pretraining approach. *arXiv preprint arXiv:1907.11692*, 2019.

[56] Zhenyu Lu, Xindong Wu, Xingquan Zhu, and Josh Bongard. Ensemble pruning via individual contribution ordering. In *Proceedings of the 16th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '10, page 871–880, New York, NY, USA, 2010. Association for Computing Machinery.

[57] Cyan Subhra Mishra, Jack Sampson, Mahmut Taylan Kandemir, and Vijaykrishnan Narayanan. Origin: Enabling on-device intelligence for human activity recognition using energy harvesting wireless sensor networks. In *2021 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1414–1419, 2021.

[58] Soo-Jin Moon, Jeffrey Helt, Yifei Yuan, Yves Bieri, Sujata Banerjee, Vyas Sekar, Wenfei Wu, Mihalis Yannakakis, and Ying Zhang. Alembic: Automated model inference for stateful network functions. In *16th USENIX Symposium on Networked Systems Design and Implementation (NSDI 19)*, pages 699–718, Boston, MA, February 2019. USENIX Association.

[59] Deepak Narayanan, Aaron Harlap, Amar Phanishayee, Vivek Seshadri, Nikhil R Devanur, Gregory R Ganger, Phillip B Gibbons, and Matei Zaharia. Pipedream: generalized pipeline parallelism for dnn training. In *Proceedings of the 27th ACM Symposium on Operating Systems Principles*, pages 1–15, 2019.

[60] Christopher Olston, Noah Fiedel, Kiril Gorovoy, Jeremiah Harmsen, Li Lao, Fangwei Li, Vinu Rajashekhar, Sukriti Ramesh, and Jordan Soyke. Tensorflow-serving: Flexible, high-performance ml serving. *arXiv preprint arXiv:1712.06139*, 2017.

[61] Nikunj C Oza. Online bagging and boosting. In *2005 IEEE international conference on systems, man and cybernetics*, volume 3, pages 2340–2345. Ieee, 2005.

[62] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. Pytorch: An imperative style, high-performance deep learning library. In *Advances in neural information processing systems*, pages 8026–8037, 2019.

[63] Heyang Qin, Syed Zawad, Yanqi Zhou, Lei Yang, Dongfang Zhao, and Feng Yan. Swift machine learning model serving scheduling: a region based reinforcement learning approach. In *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis*, pages 1–23, 2019.

[64] Xueheng Qiu, Le Zhang, Ye Ren, Ponnuthurai N Suganthan, and Gehan Amaratunga. Ensemble deep learning for regression and time series forecasting. In *2014 IEEE symposium on computational intelligence in ensemble learning (CIEL)*, pages 1–6. IEEE, 2014.

[65] Atul Rahman, Jongeun Lee, and Kiyoung Choi. Efficient fpga acceleration of convolutional neural networks using logical-3d compute array. In *2016 Design, Automation & Test in Europe Conference & Exhibition (DATE)*, pages 1393–1398. IEEE, 2016.

[66] Sara Rosenthal, Noura Farra, and Preslav Nakov. SemEval-2017 task 4: Sentiment analysis in Twitter. In *Proceedings of the 11th International Workshop on Semantic Evaluation (SemEval-2017)*, pages 502–518, Vancouver, Canada, August 2017. Association for Computational Linguistics.

[67] Victor Sanh, Lysandre Debut, Julien Chaumond, and Thomas Wolf. Distilbert, a distilled version of bert: smaller, faster, cheaper and lighter. *arXiv preprint arXiv:1910.01108*, 2019.

[68] Prateek Sharma, David Irwin, and Prashant Shenoy. Portfolio-driven resource management for transient cloud servers. *Proc. ACM Meas. Anal. Comput. Syst.*, 1(1), June 2017.

[69] Prateek Sharma, Stephen Lee, Tian Guo, David Irwin, and Prashant Shenoy. Spotcheck: Designing a derivative iaas cloud on the spot market. In *Proceedings of the Tenth European Conference on Computer Systems*, pages 1–15, 2015.

[70] Steven A Shaya, Neal Matheson, John Anthony Singarayar, Nikiforos Kollias, and Jeffrey Adam Bloom. Intelligent performance-based product recommendation system, October 5 2010. US Patent 7,809,601.

[71] Richard Socher, Yoshua Bengio, and Chris Manning. Deep learning for nlp. *Tutorial at Association of Computational Logistics (ACL)*, 2012.

[72] Richard Socher, Alex Perelygin, Jean Wu, Jason Chuang, Christopher D Manning, Andrew Y Ng, and Christopher Potts. Recursive deep models for semantic compositionality over a sentiment treebank. In *Proceedings of the 2013 conference on empirical methods in natural language processing*, pages 1631–1642, 2013.

[73] Mingxing Tan and Quoc V Le. Efficientnet: Rethinking model scaling for convolutional neural networks. *arXiv preprint arXiv:1905.11946*, 2019.

[74] P. Thinakaran, J. R. Gunasekaran, B. Sharma, M. T. Kandemir, and C. R. Das. Phoenix: A constraint-aware scheduler for heterogeneous datacenters. In *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*, June 2017.

[75] P. Thinakaran, J. R. Gunasekaran, B. Sharma, M. T. Kandemir, and C. R. Das. Kube-Knots: Resource Harvesting through Dynamic Container Orchestration in GPU-based Datacenters. In *CLUSTER*, 2019.

[76] Guido Urdaneta, Guillaume Pierre, and Maarten Van Steen. Wikipedia workload analysis for decentralized hosting. *Computer Networks*, 2009.

[77] Alexander Vezhnevets and Vladimir Vezhnevets. Modest adaboost-teaching adaboost to generalize better. In *Graphicon*, pages 987–997, 2005.

[78] Jasper A Vrugt and Bruce A Robinson. Treatment of uncertainty using ensemble methods: Comparison of sequential data assimilation and bayesian model averaging. *Water Resources Research*, 43(1), 2007.

[79] Cheng Wang, Bhuvan Urgaonkar, Neda Nasiriani, and George Kesidis. Using burstable instances in the public cloud: Why, when and how? *SIGMETRICS*, June 2017.

[80] Wei Wang, Jinyang Gao, Meihui Zhang, Sheng Wang, Gang Chen, Teck Khim Ng, Beng Chin Ooi, Jie Shao, and Moaz Reyad. Rafiki: machine learning as an analytics service system. *Proceedings of the VLDB Endowment*, 12(2):128–140, 2018.

[81] Thomas Wolf, Lysandre Debut, Victor Sanh, Julien Chaumond, Clement Delangue, Anthony Moi, Pierric Cistac, Tim Rault, Rémi Louf, Morgan Funtowicz, et al. Huggingface's transformers: State-of-

the-art natural language processing. *arXiv preprint arXiv:1910.03771*, 2019.

[82] Carole-Jean Wu, David Brooks, Kevin Chen, Douglas Chen, Sy Choudhury, Marat Dukhan, Kim Hazelwood, Eldad Isaac, Yangqing Jia, Bill Jia, et al. Machine learning at facebook: Understanding inference at the edge. In *2019 IEEE International Symposium on High Performance Computer Architecture (HPCA)*, pages 331–344. IEEE, 2019.

[83] Neeraja J. Yadwadkar, Francisco Romero, Qian Li, and Christos Kozyrakis. A case for managed and model-less inference serving. In *Proceedings of the Workshop on Hot Topics in Operating Systems*, New York, NY, USA, 2019. Association for Computing Machinery.

[84] Tien-Ju Yang, Andrew G. Howard, Bo Chen, Xiao Zhang, Alec Go, Vivienne Sze, and Hartwig Adam. Netadapt: Platform-aware neural network adaptation for mobile applications. *CoRR*, abs/1804.03230, 2018.

[85] Zhilin Yang, Zihang Dai, Yiming Yang, Jaime Carbonell, Ruslan Salakhutdinov, and Quoc V Le. Xlnet: Generalized autoregressive pretraining for language understanding. *arXiv preprint arXiv:1906.08237*, 2019.

[86] Chengliang Zhang, Minchen Yu, Wei Wang, and Feng Yan. Mark: Exploiting cloud services for cost-effective, slo-aware machine learning inference serving. In *ATC*, 2019.

[87] Honglei Zhuang, Chi Wang, and Yifan Wang. Identifying outlier arms in multi-armed bandit. In *Advances in Neural Information Processing Systems*, pages 5204–5213, 2017.

[88] Sheikh Ziauddin and Matthew N Dailey. Iris recognition performance enhancement using weighted majority voting. In *2008 15th IEEE International Conference on Image Processing*, pages 277–280. IEEE, 2008.

Appendix

A Modeling of Ensembling

While performing an ensemble it is important to be sure that we can reach the desired accuracy by combining more models. In our design, we solve our first objective function (described in Section 4.1) by combining all available models which meet the latency SLO. To be sure that the combination will give us the desired accuracy of the larger model, we try to theoretically analyse the scenario. We formulate the problem conservatively as following.

We perform an inference by ensembling 'N' models, and each of these models have accuracy 'a'. Therefore the probability of any model giving a correct classification is 'a'. We assume the output to be correct if majority of them, i.e. $\lfloor N/2 \rfloor + 1$ of them give the same result. Then, the final accuracy of this ensemble would be the probability of at least $\lfloor N/2 \rfloor + 1$ of them giving a correct result.

To we model this problem as a coin-toss problem involving N biased coins with having probability of occurrence of head to be a. Relating this to our problem, each coin represents a model, and an occurrence of head represents the model giving the correct classification. Hence, the problem boils down to find the probability of at least $\lfloor N/2 \rfloor + 1$ heads when all N coins are tossed together. This is a standard binomial distribution problem and can be solved by using the following formula:

$$P_{head} = \sum_{i=\lfloor \frac{N}{2} \rfloor + 1}^{N} \binom{N}{i} a^i \, (1-a)^{(N-i)}.$$

To further quantify, let us consider the case where we need to determine if we can reach the accuracy of NasNetLarge (82%) by combining rest of the smaller models which have lesser latency than NasNetLarge. We have 10 (therefore N = 10) such models and among them the least accurate model is MobileNetV1 (accuracy 70%, therefore a = 0.70). We need to find the probability of at least 6 of them being correct. Using the equation above we find the probability to be

$$P_{head} = \sum_{i=\lfloor \frac{10}{2} \rfloor + 1 = 6}^{10} \binom{10}{i} 0.7^i \, (1-0.7)^{(10-i)} = 0.83$$

This corresponds to an accuracy of 83%, which is greater than our required accuracy of 82%). Given all the other models have higher accuracy, the least accuracy we can expect with such an ensemble is 83%. This analysis forms the base of our ensemble technique, and hence proving the combination of multiple available models can be more accurate than the most accurate individual model.

B Why DeepARest Model?

We quantitatively justify the choice of using DeepARest by conducting a brick-by-brick comparison of the accuracy loss

when compared with other state-of-the-art prediction models used in prior work.

Table 4 shows the root mean squared error (RMSE) incurred by all the models. The ML models used in these experiments are pre-trained with 60% of the Twitter arrival trace. It is evident that the LSTM and DeepAREst have lowest RMSE value. DeepARest is 10% better than LSTM model. Since the primary contribution in *Cocktail* is to provide high accuracy and low latency predictions at cheaper cost, application developers can adapt the prediction algorithm to their needs or even plug-in their own prediction models.

C System Overheads

We characterize the system-level overheads incurred due to the design choices in *Cocktail*. The *mongodb* database is a centralized server, which resides on the head-node. We measure the overall average latency incurred due to all reads/writes in the database, which is well within 1.5ms. The DeepARest prediction model which is not in the critical decision-making path runs as a background process incurring 2.2 ms latency on average. The weighted majority voting takes 0.5ms and the model selection policy takes 0.7ms. The time taken to spawn new VM takes about 60s to 100s depending on the size of the VM instance. The time taken to choose models from the model-cache is less than 1ms. The end-to-end response time to send the image to a worker VM and get the prediction back, was dominated by about 300ms (at maximum) of payload transfer time.

D Instance configuration and Pricing

Instance	vCPUs	Memory	Price
C5a.xlarge	4	8 GiB	$0.154
C5a.2xlarge	8	16 GiB	$0.308
C5a.4xlarge	16	32 GiB	$0.616
C5a.8xlarge	32	64 GiB	$1.232

Table 7: Configuration and Pricing for EC2 C5 instances.

E CIFAR-100 and BERT Models

Table 8 shows the different models available for image prediction, that are pretrained on Keras using `CIFAR-100` dataset.

Model	Params (M)	Top-1 Accuracy(%)	Latency (ms)	P_f
Albert-base [51]	11	91.4	55	7
CodeBert [32]	125	89	79	6
DistilBert [67]	66	90.6	92	5
Albert-large	17	92.5	120	4
XLNet [85]	110	94.6	165	3
Bert [30]	110	92	185	3
Roberta [55]	355	94.3	200	2
Albert-xlarge	58	93.8	220	1
Albert-xxlarge	223	95.9	350	1

Table 9: Pretrained models for Sentiment Analysis using BERT.

Similarly Table 9 shows the different models trained for BERT-based sentiment analysis on twitter dataset.

Model	Params (M)	Top1 Accuracy %	Latency (ms)	Pf
Squeezenet	4,253,864	70.10	43.45	10
MobileNEt V2	4,253,864	68.20	41.5	10
Inception V4	23,851,784	76.74	74	6
Resnet50	95,154,159	79.20	98.22	5
ResNet18	44,964,665	76.26	35	6
DenseNet-201	20,242,984	79.80	152.21	2
DenseNet-121	8,062,504	78.72	102.35	3
Xception	22,910,480	77.80	119.2	4
NasNet	5,326,716	77.90	120	3
InceptionResnetV2	2,510,000	80.30	251.96	1

Table 8: Pretrained models for CIFAR-100 using Imagenet.

F Spot Instance Price Variation

We profile the spot price of 4 types of `C5` EC2 VMs over a 2-week period in August 2020. The price variation is shown in Fig18.

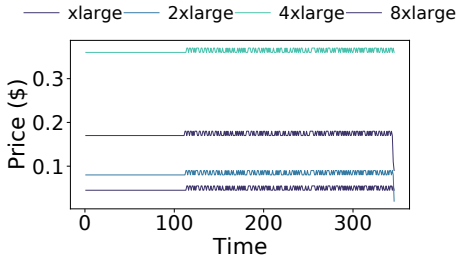


Figure 18: Spot instance price variation (time is in hours).