DIAS: Distributed Inference as a Service

Suleman Ahmad, Vinay Banakar, Amos Kendall

5/2/2021

1 Introduction

While much of the focus of machine learning research is on the process of training models (i.e., learning) there are a unique set of distributed system challenges around the process of serving and deploying those models that are often overlooked. For example, machine learning training can often be done in an offline setting where latency and scalability are not significant factors. Online inference serving, on the other hand, is latency sensitive and must scale to meet the demands of a growing workload. Additionally, as the number of different machine learning applications grows we also need to have flexible frameworks that can ease model deployment, while improving throughput, accuracy, and robustness without modifying the underlying machine learning models. To address these challenges, we present DIAS, Distributed Inference as a Service, a modular architecture system that reduces the complexity of implementing a prediction serving stack and achieves four crucial properties of a prediction serving system: low latencies, high reliability, improved accuracy, and scalability. DIAS is a framework that distributes multiple machine learning models across a back-end cluster, and exploit ensemble techniques to robustly select and combine predictions from these models for better inference accuracy. We also studied the trade off between cost, performance and accuracy of deploying DIAS on two different cloud platforms. Our system is open-source and can be found at [1].

2 Related Work

The approach we have taken for DIAS is very similar to Clipper [2], TensorFlow Serving [3], and Velox [4] model inference systems. Clipper provides a layered architecture system that reduces the complexity of implementing a prediction serving stack and isolates end-user applications from the variability and diversity in machine learning frameworks by providing a common prediction interface. One of its most notable features is the adaptive batching mechanism, where a queue associated to each model that is tuned to the latency profile of that model's container, which greatly improves prediction throughput. TensorFlow Serving is the open-source prediction serving system developed by Google specifically for TensorFlow models. It does not employ caching but instead leverages batching and hardware acceleration to improve prediction performance. While Velox is a UC Berkeley research project to study personalized prediction serving with Apache Spark. It utilizes caching of prediction responses at various levels in their systems, and also uses a straggler mitigation strategy to address slow feature evaluation. All three systems propose mechanisms for improving prediction latency and throughput of machine learning inference workloads.

We have designed DIAS, a general purpose prediction service, that is agnostic to the deployment platform and can easily be integrated on multiple cloud architectures, as opposed to the previously described systems. DIAS supports a wide range of machine learning models and frameworks and simultaneously addresses latency, throughput, and accuracy better than a single serving system.

Ensemble Majority Voting: It is a well-known result in machine learning [5, 6, 7] that prediction accuracy can be improved by combining predictions from multiple models. For example, bootstrap aggregation [8] is used widely to reduce variance and thereby improve generalization performance. Ensembles have also been used to win the Netflix challenge [9], and a carefully crafted ensemble of deep neural networks was used to achieve state-of-the-art accuracy on the speech recognition corpus Google uses to power their acoustic models [6]. DIAS employs ensemble based strategy by carrying our majority voting on the responses returned by the deployed models. It seeks to improve performance by considering predictions from all available models rather than relying on individual responses.

Prediction Serving: There has been considerable prior work in model and application specific prediction-serving. Much of this work has focused on content recommendation, including ad-targeting [10], or product recommendations [11]. There has also been recent success in speech recognition [12] and internet-scale resource allocation for improved video streaming [13]. While many of these applications require real-time predictions, the solutions described are highly application

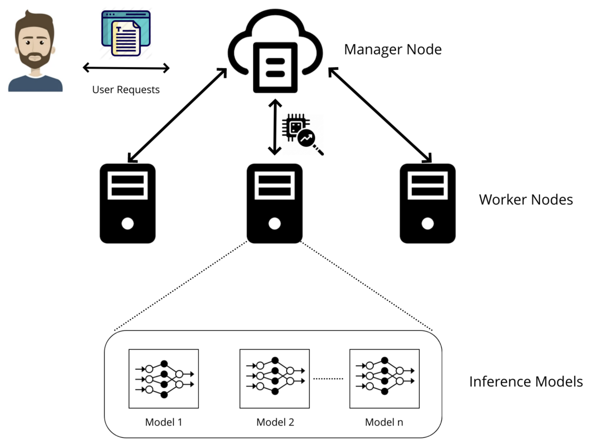


Figure 1: DIAS Architecture

specific and tightly coupled to the model and workload characteristics. As a consequence, much of this work solves the same systems challenges in different application areas. DIAS is a general-purpose system that can help serve many of these applications, as it is flexible enough to operate on any type of the deployed models.

3 Design

Machine learning models are very compute intensive, therefore their deployment on a single node can become a bottleneck when multiple clients try to use it as a prediction service. We designed DIAS, a framework to overcome such limitations by deploying and managing models over a cluster of machines. Our framework can be divided into two independently scalable components: `Manager` module and the `Worker` engine. Figure 1 illustrates the architecture of the system. The Manager is responsible for handling user requests, forwarding the processed user input to the back-end Worker nodes, and aggregating predictions for ensemble voting [14]. Each model is replicated on every Worker node, allowing each Worker to choose any of the locally available models for inference. The system is extensible and supports multiple inference model types, multiple input types, and allows for adaptive load balancing by redistributing load on unused model replicas across Worker nodes.

3.1 Manager

The Manager module acts as a leader in the system, as it exposes endpoints for all user interactions, and consequently, propagates inference requests to the back-end Worker nodes. We have followed an architecture design similar to Primary-Backup approach [15], but our Manager module is also replicated, and scales as the load on the system increases (depending on the choice of the deployment platform). The Manager stores a unique tag for each of the deployed ML model. Each time a new model is made available for inference, it must be registered with the Manager. No client state is stored which allows the system to easily spawn new Manager nodes upon failure or when balancing excessive load.

For ease of use and simplicity from a user's perspective we opted to use RESTful implementation at the Manager. The following API endpoints are publicly exposed:

- **\predict** Accepts POST requests with the following mandatory JSON fields: 1) *Input*: This is a *serialized object* on which inference needs to be computed, 2) *Category*: Specifies the type or category of the input, and 3) *Task*: Defines the type of inference engine to run for that category. This endpoint is for computing inference on a single instance.

- **\batch_predict** Similar to the \predict endpoint, but accepts nested JSON fields for multiple *Inputs*, for the same *Category* and *Task*. Batching input for ML models improves prediction performance.

- **\help** A GET request endpoint that provides detailed instructions to users about the type of supported back-end *Categories* and *Tasks*.

All incoming user requests are parsed by the Manager. For each request *Category* and *Task* fields are extracted to determine the set of ML models that could perform inference for the supplied *Input*. For each *capable* ML model, which is identified by its unique tag, the Manager forwards the inference details to a Worker node by making asynchronous requests. This makes sure that an inference request is only made to the models that can handle that type of input, and also allows for parallel computation at each of Worker nodes. This greatly improves prediction latency over an ensemble of models.

The Manager takes the responsibility of aggregating all the prediction results returned by the Worker nodes, and performs majority voting on the collected results. Worker nodes that return errors are ignored from the voting procedure. Ensemble majority voting allows combining predictions from multiple models thus improving overall accuracy as compared to the result of any single model. This technique of course relies on the notion that the *majority* of the deployed models are "robust" to receive good performance results. An average of the confidence score for each of the models is also taken, and is sent back to the user along with the prediction result.

The prediction API endpoints support an optional parameter of *Model Count*, which is used for limiting the ensemble size used for prediction. This makes sure that the Manager only waits for responses for the first N models, and does an ensemble majority voting only on them. This is useful when prediction latency is more critical than the overall accuracy as straggler models are not taken into consideration which improves tail latency.

3.2 Worker

Workers are responsible for running the core inference engine of the service. Every Worker node runs a thin layer of inference interface, that processes requests from the Manager and sends the results back. The Worker engine interface is stateless and flexible as it can run any of the models that are persisted locally on the Worker nodes. As long as at least one Worker node is running, the service would always be available.

Upon receiving a prediction request from the Manager, the interface forks a new process which uses the unique model tag forwarded by the Manager to locate the appropriate model files. Model files are stored on disk and are also cached in memory once they are used. The model files include a data transformation function as well as a serialized version of the model including the weights and relevant hyperparameters. Once the model files are read, the prediction *Input* is then de-serialized, transformed via the data transformation function and then passed to the respective model for generating prediction. Once the results are available, they are sent back to the Manager. Upon any errors, such as invalid *Input* type, the Worker sends back a detailed log of the error to the Manager (which are made part of Manager logs).

Currently, our system requires manual placement of ML models on the Worker nodes. Additionally, we expect the DIAS service provider to already have trained ML models. Ability to automatically deploy ML models is part of our future work.

4 Deployment

One of our design goals was to structure our framework in such a way that its deployment is not dependent on a particular platform. We were able to achieve most of this flexibility by containerizing and breaking down our application into two independent sets of stateless components i.e. the Manager and Worker. This gave us the freedom to evaluate and deploy our system on multiple platforms. We initially deployed our system as a single node dockerized application on a Cloudlab machine, and then expanded its deployment to Google Kubernetes Engine [16] and Google Functions [17] distributed platforms.

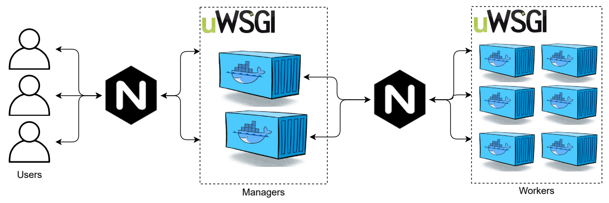


Figure 2: Single node DIAS architecture

4.1 Local Dockerized Deployment

We started off by deploying our containerized framework on a single node. This involved running Manager and Worker docker containers on the same local machine, and calling all our REST APIs locally. This allowed us to observe and verify behavior without considering additional cloud computing infrastructure overhead. In particular, as we detail in Section 6.1.1, we were able to use this local deployment to capture the overhead of the REST APIs and containerization.

We also investigated Docker's ability to leverage uWSGI [18] for scaling. In particular, uWSGI allows the docker container to scale up the number of processes running to respond to REST APIs. The main goal here was to scale up the Worker container itself to respond to the Manager's asynchronous requests in parallel. We use a Nginx [19] server as a web proxy for Managers and Workers, this provides the service like abstraction for clients (when they use Manager) and Managers (when they use Workers). Figure 2 depicts our single node architecture.

4.2 Google Kubernetes Engine (GKE)

In our single node setup we grew tired of manually configuring the number of docker containers required as the number of concurrent users changed. Moreover, the earlier design did not provide any fault tolerance guarantees. This motivated us to consider container orchestration frameworks such as Kuberenetes (K8s) [20] for our deployment. GKE [16] provides secured and managed K8s cluster services for users in public cloud. Each cluster contains nodes (physical server) and each node contains pods (smallest deployable computing unit). Each pod constitutes one or more containers that share storage and network resources. K8s maintains the specified number of replicas by monitoring pods and registering events in etcd [21]. It also provides features to load balance requests among pods based on various metrics and allows containers to auto scale pods when a threshold is met.

For DIAS, each pod constitutes Nginx, uWSGI and the respective type of containers. The images for these containers are stored in a global cloud registry after be-

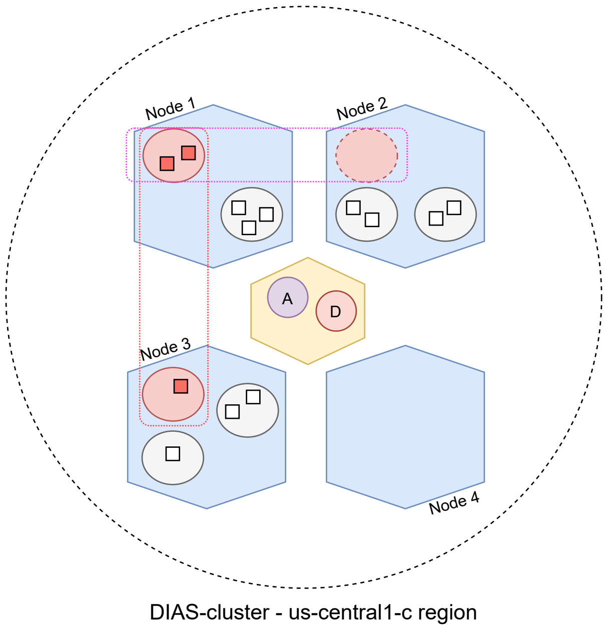


Figure 3: K8s DIAS architecture: Yellow hex is the control plane which consists of the rules for the cluster deployment $\{D\}$ and for autoscale configuration $\{A\}$. Blue hex represents the physical nodes, each node has pods marked in circles. Red pods are Managers and grey are Workers. Each pods have containers, nginx and uWSGI. Node 4 is shown empty to illustrate that K8s bin pack pods, so even if node 4 is at 0% utilization, no pods are scheduled on it.

ing built locally. The Worker image is generally large since we opted to package the models with the image rather than hosting the models separately on a shared storage. This was a trade off between copying models after deploying the container (faster start time) or having a heavier image (slower start time) with no copying. Having a two level of abstraction allowed us to actively scale the number of pods and number of containers per pods to suite the load requirement. A set of pods are called a `Deployment` kind and each of these deployment had a `Service` kind associated with it. The service for Manager pods is of type `LoadBalancer` which distributes the requests among these pods. The service for Worker pods is of type `NodePort` which also distributes requests but does not provide an public external IP to the service unlike LoadBalancer kind. Manager deployment was configured to maintain at least two replicas and Worker to have three replicas all the time. This was to ensure our service was available during arbitrary failures.

Figure 3 provides a high level design of our setup. Since our workloads are compute bound, we chose to use cpu utilization as the trigger to scale the number of pods. We configured both Manager and Worker pods to scale up to a max of 50 instances when CPU utilization per pod reached 70% utilization. The minimum number of pods are dictated by the replicas set during deployment (which is 2 or 3). The cluster is made of four `e2-standard-4` machines which consists of 16 GB memory and 4 vCPUs each.

4.3 Google Functions

We observed that our design is implicitly event driven and deploying a large number of nodes in a cluster is wasteful. At the same time, we did not want to provide bad service to our clients. Thus, we retrofitted our existing application to be fully functional using serverless. We used Google functions to deploy an instance of Manager and Worker. This was an easy transition since they were designed to be stateless. Worker function however can't be initialized with the models packed together. Hence we stored our models on an object storage bucket. So at startup the Worker function only copies the model it is interested in to a local temporary directory. The serverless function can then load the model files from storage during startup. Deploying on google functions was easy since our framework is modular and the independent set of components are stateless.

On deployment, we did not have to worry about the metrics to consider for scaling since load balancing was provided by Google cloud by default. We only had to specify the max number of function instances that can be deployed at once. This was set to 50 for both Managers and Workers in DIAS, a total of 100 at max. Figure 4 shows the event driven design of DIAS-serverless.

5 Case Study: Fake News Detection

To develop and test DIAS, we use a specific machine learning problem as a case study. The correctness of DIAS is tied closely to the performance of the machine learning models we use for inference serving. That is, DIAS can be considered to perform correctly if it serves the expected model inference for each model and correctly selects the majority vote. For that reason, we provide an overview of the inference task and the machine learning models we deploy in the remainder of this section.

5.1 Problem Description

For our case study, we chose the problem of fake news detection. Fake news has been an increasingly important topic, as political forces seek to use new technologies to disseminate unverified knowledge and manipulate public

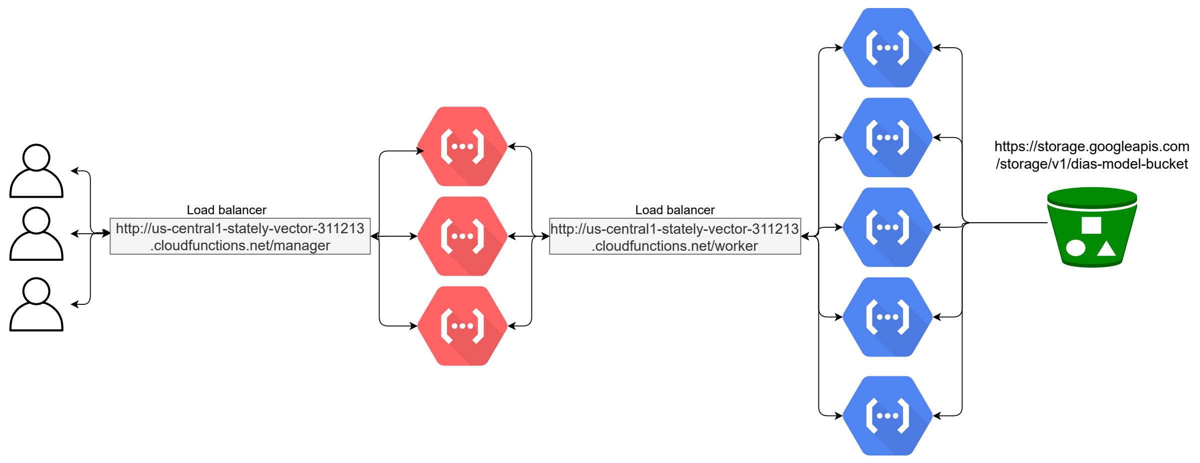

Load balancer
http://us-central1-stately-vector-311213
.cloudfunctions.net/manager
Load balancer
http://us-central1-stately-vector-311213
.cloudfunctions.net/worker
https://storage.googleapis.com
/storage/v1/dias-model-bucket

Figure 4: Serverless DIAS architecture: Red instances are Managers and Blue are Workers. Green bucket contains all the models. Each Worker downloads the model it is interested in.

opinion towards some specific agenda. For example, in the presidential election in 2016, various kinds of fake news about the candidates were widely spread through both official news media and the online social networks in an attempt to sway the outcome of the U.S. Presidential election [22]. This trend has prompted large internet companies like Google and Twitter to vow to block fake news on their platforms [23]. This stated goal naturally leads to the drive to automate fake news detection instead of relying on human fact checkers. Natural Language Processing (NLP) is an obvious approach to try to understand if there are structural differences between fake and true news.

More technically, we treat this problem a binary classification supervised learning problem where the input is raw text, and the output is a classification of fake or true. We used the T1-CNN dataset built in [24] to train and evaluate our models for fake news detection. It contains 20,015 samples of news articles scraped from over 240 different websites including well-known sources such as The New York Times or The Washington Post. The dataset was collected by the authors at the time of 2016 American presidential elections therefore many articles refer to election news. Each data sample was manually annotated as either fake (0) or true (1). There are 12,000 fake articles and 8,015 true articles. For each article, the dataset contains several features like titles, body content, publisher details, authors, images and URLs. However, we only used the article text body in order to focus explicitly on the semantic content of the document. Since the dataset has been peer-reviewed and published, we believe it is of sufficient quality to train machine learning models on it.

5.2 Model Overview

One of the promises of DIAS is to use ensemble techniques to provide increased accuracy to the user. In order to validate this assumption, we trained and deployed five different models.

5.2.1 Feature Engineering

Most machine learning tasks require numeric inputs. Since our fake news case study needs to start with raw text, we need to extract a numeric representation of the text before inference (and training). We calculated the Term Frequency-Inverse Document Frequency(TF-IDF) scores for each of the text input. TF-IDF provides a statistical measure to see how important a word is for a document. It gives frequency score to words by highlighting the ones which occur more frequently. Using the TF-IDF, we tokenized the text input, and encoded them into inverse document frequency scores. Since the size of TF-IDF vectors is equal to the total dictionary of words used across all articles in the training set, we applied Singular Value Decomposition (SVD) to the TF-IDF sparse matrix to create a 100-dimensional feature vector for each document. This decomposition allows us to work in a more new, tractable feature space while still preserving the strongest signals of the original feature space.

5.2.2 Logistic Regression

Logistic Regression takes the dot product of a weight vector with a feature vector and compresses this continuous output into binary class probability using the *softmax* function. It uses gradient decent to find the weight vector that minimizes a given loss function across all training examples. For our model, we used the standard L_2 loss function.

5.2.3 Support Vector Machine

Support Vector Machine (SVM) is a supervised machine learning model, that is used for both classification and regression tasks. Under the context of classification, SVM finds an optimal hyperplane that can separate data points of two different classes in the feature space. Its objective is to find a plane that has the maximum margin, i.e the maximum distance between data points of both classes, so that future test time data points can be classified with more confidence. SVM uses the kernel trick to map inputs into a high dimension features space, enabling it to draw non-linear decision boundaries in the input space.

5.2.4 Random Forest

A random forest is an ensemble collection of decision trees. Each decision tree builds a binary tree of nodes where each node represents some criteria on the feature space of the training set. If a sample meets the criteria at a given node, it moves to the left child, if it does not meet the criteria, it moves to the right child. Thus, any single sample of the training set can trace a path from the root of the tree to a leaf node based on the sets of criteria that is encounters at each node. Each leaf node is labeled, leading to a classification of the input. The goal of each node is to evenly partition the training set based on some value of that feature in order to minimize the size of the tree. Each tree uses some randomness when picking the options for the condition criteria at each node, leading to a unique tree. The entire forest uses the average response across all trees to make a prediction, which helps avoid over-fitting. We use 100 decision trees in our random forest.

5.2.5 AdaBoost

Similar to a Random Forest, AdaBoost uses an ensemble of decision trees for classification. However, instead of a simple average response, it uses a sequential approach. It starts with a single decision tree, and then identifies any samples that are misclassified by this initial tree. Subsequent trees are trained by weighting the miscalsified samples of the previous tree more heavily, causes the subsequent trees to do a better job fitting these specific samples. Thus, each layer of the AdaBoost classifier is tuned to correct the mistakes of previous layers. The goal is to nest all of these layers together to account for all samples in the training set.

5.2.6 Multi-Layer Perceptron

A multi-layer perceptron is a simple form of a neural network. Ours consists of 100 hidden nodes, and 1 output node. Each node, takes a weighted sum of inputs, and uses the relu function to create a non-linear output. The output is then used as input in the next layer. The final output node uses the softmax function to map the continuous output to a binary classification confidence score. The weights of the model are updated uses back propagation and gradient descent.

Cloudlab (`c240g2`)	
CPU	2x Intel(R) Xeon E5-2660 v3 2.20GHz 10-core
Memory	157GB DDR4 RDIMMs @ 2133 MT/s
Storage	447GB SATA SSD
OS	Ubuntu 20.04 LTS

Table 1: Local Experiments Machine Specification

6 Evaluation

6.1 Dockerized Performance (AMOS)

To better understand the behavior of our containerized environment, we ran three different local dockerized experiments each with a different configuration. For these experiments we used a Cloudlab machine, whose specifications are provided in Table 1.

6.1.1 Experiments

For each experiment, we loop through our entire dataset and evaluate our logistic regression model against each sample. We track the latency of each request and report the median and 99% tail. We also calculate the total operations per second. The results can be seen in figures 5 and 6. For our first experiment, we run inference on our logistic regression model without any containers (No Container). That is, we simply run a python script using the python interpreter. For our second experiment (Worker Container), we run the Worker container locally and call directly into it using the REST API, bypassing the Manager container. For the last experiment, we run both Manager and Worker locally, and call into the Manager container using the relevant REST API. For each experiment, the same inference workload is being performed. Thus, we are able to observe the overhead from the REST API calls and docker configuration.

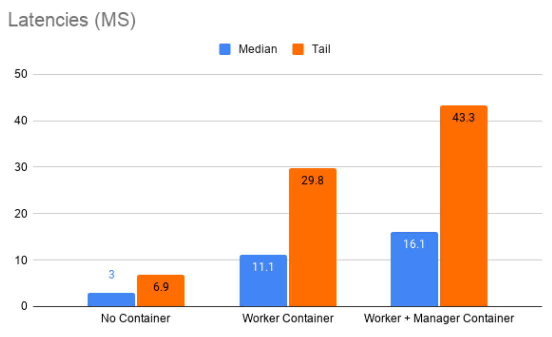


Figure 5: Local configuration latencies

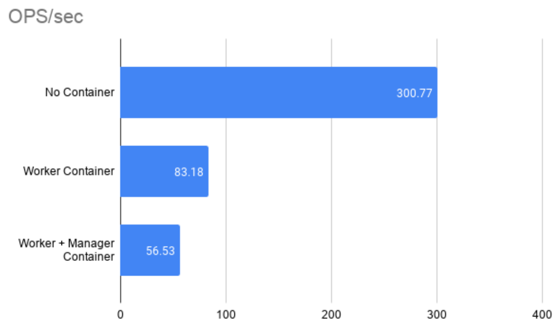


Figure 6: Local configuration throughput

6.1.2 Results

It is worth noting that even without any containers, the tail latency is significant. This is because the dataset contains articles of varying lengths. Larger articles lead to slower preprocessing and inference. Since the entire raw text is sent as a payload during each REST API call, this large article effect is magnified with more total API calls.

As expected, adding containers and the REST API calls adds overhead. We see that each additional REST API adds between 5 and 8 ms to the median latency of a request. This results in a 5x slowdown between the No Container and the All Container approaches. However, one of the main advantages of DIAS is the ability to run inference in parallel on replicated Worker containers. In our case study, we deployed five models, so each call to the Manager runs 5x as many model inference queries. Compared to a serial computation, this parallelization gives us close to a 5x speedup. Extrapolating these results, we can infer that the back end inference throughput of the clustered environment is similar to a serialized local deployment. Indeed, we see in later experiments that running all 5 models in our cloud computing environment has a similar throughput to the fully containerized local experiment. Furthermore, as the number of models being used increases, this parallelization becomes even more significant, dominating serial model computation.

6.2 Kubernetes Deployment Performance

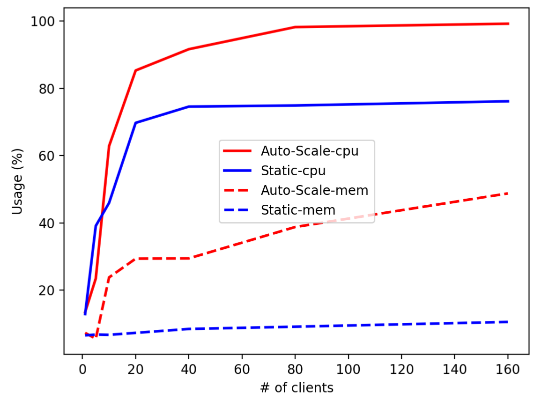
As described in Section 4.2 our cluster is set up at us-central-1c (Iowa) region. We picked this since clients were hosted on a Wisconsin Cloudlab machine (`c240g2`), we wanted to keep the network latency to a minimum. Figure 7(a) shows the average resource usage of all the nodes serving requests. This confirms that our service is highly CPU bound and illustrates that even with increased number of active pods the average node CPU usage will increase since more pods per node are allocated. With static configuration the CPU utilization is capped at 70% because new pods can't be spawned on remaining nodes to increase the overall CPU usage.

Figure 7(b) shows the gradual growth of the number of pods up to 100. This shows that our autoscaling rules work as expected. Both Manager and Worker service were configured to scale up to 50 pods when they reach 70% CPU utilization. This takes care of automatically queuing requests at Manager or spawning more Workers relative to the Manager to alleviate the load. Pods once deployed do not get rescheduled to nodes that are underutilized, so for smaller number of concurrent requests, although multiple Workers compete for CPU utilization of a single node (pushing it to 100%) a neighbouring node could be at 0% with no pods deployed on it. This seems to be a K8s feature or an issue. Dynamic relocation of pods on different nodes based on node resource usage would be helpful.

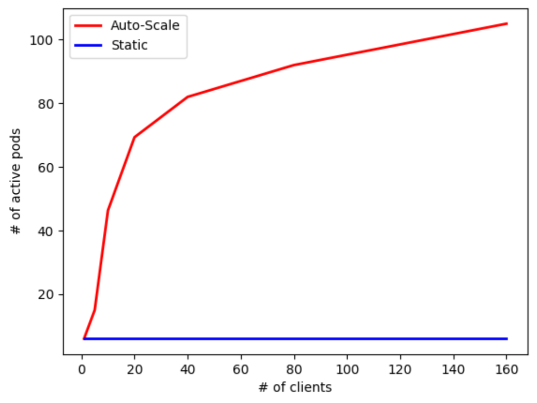
Figure 8 is an interesting graph because it demonstrates that we can offer DIAS to a large number of clients without hurting latency of each request. We can scale linearly due to auto scaling pods and effectively using the available cluster capacity. Although 160 concurrent clients seem small, each client makes about 20 thousand prediction requests each of size varying from a few bytes to a few KBs. The static line in the graph shows how with limited cluster capacity (mimicked by limiting number of pods) we can only support 20 concurrent clients. After which the bandwidth stalls and per client latency increases, providing a bad experience to the user. Overall, we recorded an aggregate bandwidth of 4 MB/s for 160 clients. Since this graph is linear we expect to be no where close to actual maximum throughput possible. We ran out of promotional credits to continue our experiments with larger number of concurrent clients.

6.3 Cloud Functions Performance

DIAS serverless setup is described in Section 4.3. Unlike our Kubernetes experiments we did not have access to CPU utilization of the instances deployed to run our



(a) Average node CPU and memory usage



(b) Autoscaling pods with concurrent requests

Figure 7: Kuberenetes resource usage and pod scaling for up to 160 clients.

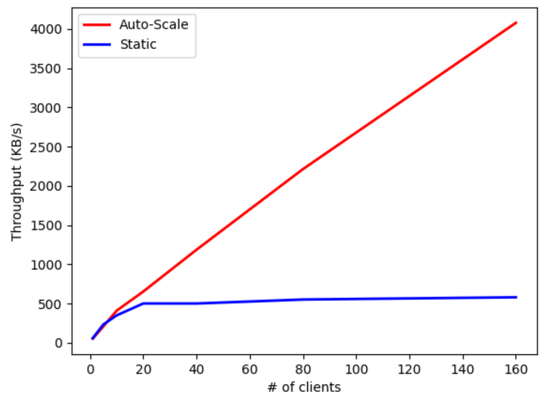


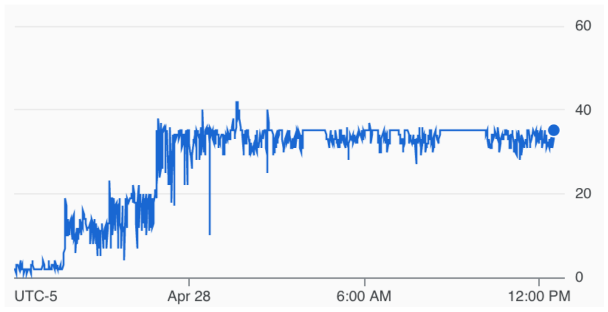
Figure 8: Total throughput achieved between static and scaling setup using K8s

functions. We could have deployed measurement functions and potentially extrapolate it from `/sys` as demonstrated by Wang et al. [25]. However, we focused on running the experiments with the limited time we had. Due to limited resources left we had to cap the total number of function instances to 70. As seen in Figure 10, this had a huge impact on performance. After 80 concurrent clients the throughput decreases due to congestion. The more interesting part is the poor performance of serverless for smaller concurrent requests. For 40 clients we could only achieve 750KB/s. This is surprising because we do not expect the system to show congestion until after 80 clients. However, we saw K8s providing 1250 KB/s for 40 clients and 2000 KB/s for 80 clients which is about 2.4x higher than serverless.

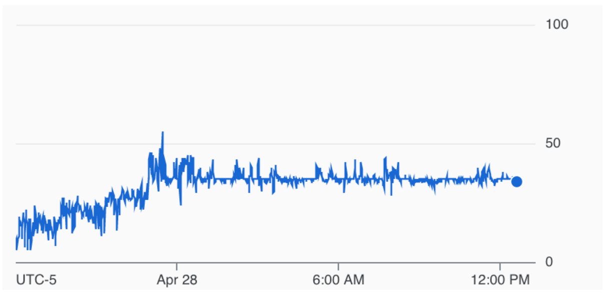
We suspect this to be the cost of higher startup time required by functions, but have not been able to verify this hypothesis. Figure 9 shows the total instances deployed during our performance experiments. The number of instances are capped to 35 for manager and worker each. However we observe the total number of instances growing beyond this cap for both. The initial few minutes indicate just 1 client sending 20k requests in sequence synchronously. As expected the manager has a small number of instances, but during the same period there are a larger number of workers running. This shows the workers are the bottleneck; There is no queuing of requests for a single manager, but rather new workers are spawned to handle the stream of requests from managers. Each function is deployed with 1GB of memory and we verified that for either functions memory was not a bottleneck. The manager on an average used 70MB and workers used 380MB per function execution.

6.4 Cost Analysis

Both distributed platforms provided us with many useful features such as fault-tolerance, dynamic load balancing, and elastic compute environment, but they are not cheap to operate. While running our framework over each of the platforms, we quickly ran out of 300$ worth of credit in a week's time period. Google Kubernetes deployment based its pricing on the provisioning of resources that were being used by the pods. Therefore, as long as the pods were running (although idle) we were being charged for their execution. Dynamic scaling of the pods also incurred extra cost. For Google Functions we saw that we were only priced per incoming inference request. Whenever a new function was invoked to carry out the prediction, the cost would fluctuate depending on the resources used by the invoked function. Never-



(a) Managers



(b) Workers

Figure 9: Active instances for 1 to 160 clients, with number of clients doubling after reach round.

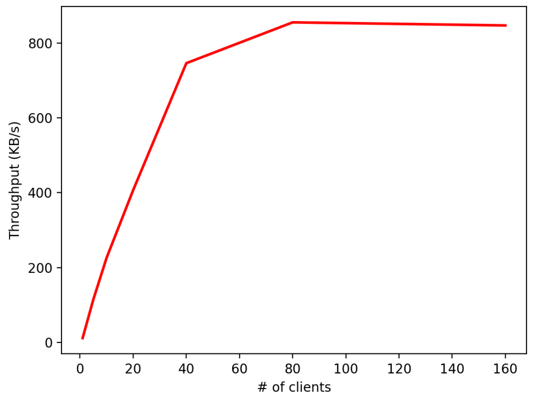


Figure 10: Total throughput achieved using Google functions with maximum 70 active active instances.

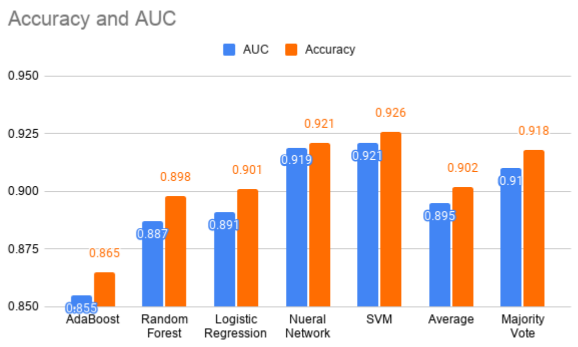


Figure 11: Model performance

theless, logging service that provide features like store, search, analyze, monitor and alert on logging data and events costed us the most. With 0.5$/GB and our logging data amounting to 300GB we inadvertently spent $150 for it. This is auto enabled in google cloud and total log data is the sum of multiple runs on K8s and google functions. These types of hidden costs are a danger of using cloud computing.

6.5 Ensemble Evaluation

We held out 20% of our dataset from training to be used for evaluating model performance. We use both accuracy and AUC metrics for measuring performance. Accuracy represents the portion of test samples that are classified correctly. AUC refers to the Area Under the Curve for the Receiver Operating Characteristic (ROC). The ROC plots the True Positive vs. False Positive rates for varying levels of discrimination thresholds for the classification confidence boundary. The area under this curve represents an accuracy measure that accounts for the skew in the dataset. We show the results of each model against this test set in Figure 11.

Since our dataset has more false samples than true samples, we see that models tend to do well in regards to accuracy by favoring the label of fake news. This is clearly seen because the AUC is lower for all models than the accuracy. Models like SVM and our Neural Network are less skewed and therefore more trustworthy.

One disappointing result is that the Majority Vote is less accurate than our most accurate single model (SVM). One hypothesis for this result is that we use the same feature space as input for all of the models. In this sense, all the models have a similar perspective on the data, so there is no advantage to considering the information provided by the less accurate models.

That being said, it is important to note that even with several less accurate models, the Majority Vote clearly outperforms the Average, showing that it goes a long way towards making up for the less accurate models.

7 Future Work

We evaluated the primary goals of DIAS and demonstrated that it can scale well for varying request size and concurrent requests. However, we believe DIAS can grow beyond an inference service. In the future, we would like to allow users to upload new models and update existing ones. This will help small startups to scale rapidly without having to worry about model distribution or infrastructure management. Furthermore, DIAS can provide a marketplace of models, which users can use for development, testing and perhaps as a client facing service as well. These models can be rented or bought by users as a single service that DIAS provides. Uploaded models and the data can be version controlled to revert back to predictable results. We would like to also support strong authentication and privacy features later. Furthermore, similar to [2] we would also consider adding adaptive batching of requests for better prediction-serving throughput.

References

[1] Suleman Ahmad, Vinay Banakar, and Amos Kendall. DIAS. `https://github.com/apkendall10/DIAS`.

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

[3] TensorFlow Serving. `https://www.tensorflow.org/tfx/serving/serving_basic`, 2021.

[4] 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.

[5] Tianqi Chen and Carlos Guestrin. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*, pages 785–794, 2016.

[6] Geoffrey Hinton, Oriol Vinyals, and Jeff Dean. Distilling the knowledge in a neural network. *arXiv preprint arXiv:1503.02531*, 2015.

[7] Kevin P Murphy. *Machine learning: a probabilistic perspective*. MIT press, 2012.

[8] Leo Breiman. Bagging predictors. *Machine learning*, 24(2):123–140, 1996.

[9] Joseph Sill, Gábor Takács, Lester Mackey, and David Lin. Feature-weighted linear stacking. *arXiv preprint arXiv:0911.0460*, 2009.

[10] H Brendan McMahan, Gary Holt, David Sculley, Michael Young, Dietmar Ebner, Julian Grady, Lan Nie, Todd Phillips, Eugene Davydov, Daniel Golovin, et al. Ad click prediction: a view from the trenches. In *Proceedings of the 19th ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 1222–1230, 2013.

[11] 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.

[12] Apple Siri. `https://www.apple.com/siri/`, 2021.

[13] Aditya Ganjam, Faisal Siddiqui, Jibin Zhan, Xi Liu, Ion Stoica, Junchen Jiang, Vyas Sekar, and Hui Zhang. C3: Internet-scale control plane for video quality optimization. In *12th {USENIX} Symposium on Networked Systems Design and Implementation ({NSDI} 15)*, pages 131–144, 2015.

[14] Thomas G. Dietterich. Ensemble methods in machine learning. In *Multiple Classifier Systems*, pages 1–15, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[15] Navin Budhiraja, Keith Marzullo, Fred B Schneider, and Sam Toueg. The primary-backup approach. *Distributed systems*, 2:199–216, 1993.

[16] Google Kubernetes Engine - Google Cloud. `https://cloud.google.com/kubernetes-engine/docs`, 2021.

[17] Google Functions - Google Cloud. `https://cloud.google.com/functions/docs`, 2021.

[18] uWSGI Project. `https://uwsgi-docs.readthedocs.io/en/latest/`, 2021.

[19] Nginx webserver. `https://nginx.org/en/docs/`, 2021.

[20] David Bernstein. Containers and cloud: From lxc to docker to kubernetes. *IEEE Cloud Computing*, 1(3):81–84, 2014.

[21] etcd. `https://etcd.io/`, 2021.

[22] Hunt Allcott and Matthew Gentzkow. Social media and fake news in the 2016 election. *Journal of Economic Perspectives*, 31(2):211–36, May 2017.

[23] Google and twitter vow to block voting misinformation. *BBC News*, September 2021.

[24] Yang Yang, Lei Zheng, Jiawei Zhang, Qingcai Cui, Zhoujun Li, and Philip S. Yu. Ti-cnn: Convolutional neural networks for fake news detection, 2018.

[25] Liang Wang, Mengyuan Li, Yinqian Zhang, Thomas Ristenpart, and Michael Swift. Peeking behind the curtains of serverless platforms. In *2018 USENIX Annual Technical Conference (USENIX ATC 18)*, pages 133–146, Boston, MA, July 2018. USENIX Association.

[26] gRPC. `https://grpc.io/`, 2021.