Gradient Boosted Decision Tree Inference for Intelligent Load Balancing of Web Requests

Connor Rawls
C00211180@louisiana.edu
University of Louisiana, Lafayette, Louisiana,

***Abstract*—Web requests and RESTfull services represent a prominent ecosystem in the world of client/server architectures. These systems are used from a variety of applications such as in simple websites to the surging topic of microservices. Specifically, these microservice environments are user-facing and must adequately handle large influxes of requests, lest the Quality of Service will be infringed upon. Contemporary load balancing algorithms utilize heuristic approaches to maximize the performance of these web systems. This traditional approach requires significant knowledge on the entire system. Additionally, these algorithms may be prone to human error. With this fact in mind, there lies a possibility for exploring nontraditional approaches to load balancing. Machine learning algorithms have been proven to accurately compensate for unknown variables in system state. Embedding an ML model into a load balancing algorithm for web requests can abstract away the necessity for preemptive algorithm design, as well as provide the ability to adjust to fluctuating workloads properly. Utilizing Gradient Boosted Decision Tree models, a load balancing algorithm is provided system metrics it otherwise was agnostic towards. The ML-ingrained system proved to provide possible benefits in load balancing under certain scenarios. Specifically, in one range, the ML load balancer showed a 69.91% decrease in error rate as compared to a representable default load balancer.**

***Index Terms*—Gradient Boosting, Decision Trees, Load Balancing**

I. Introduction

CLOUD environments provide a dominant means to deploy applications in the realm of web services today. In particular, the cloud is very suitable for applications that are prone to drastic fluctuations in resource consumption. Many web services can be considered user-facing. As users sometimes exhibit unpredictable behaviors, such as those that can be seen with large, sudden influxes in online traffic, these web services must properly adjust to bursts in workload. If these services fail to handle these bursts, the service's Quality of Service will not be met, leading to decreased user satisfaction and, in worst-case scenarios, system failure.

One benefit that comes with deploying applications in the cloud is in its elastic nature. Cloud elasticity allows applications to seamlessly obtain the additional resources it demands in the case of increased user workloads, maintaining satisfactory response times. Oppositely, applications can just as easily relinquish resources to reduce their power and cost consumption, keeping operation costs at a minimum.

This scaling of resources comes with a penalty, however. Increasing system resources for larger compute power or data capacity requires more energy from the cloud providers perspective. In turn, major expenditures are brought forth by increasing a cloud application's resources. In the realm of bursty behavior, applications may see large, unexpected spikes in operating costs if the acquisition of resources is allowed to run rampant. Therefore, there lies a delicate balance between maximizing the performance of the application while minimizing its cloud operating costs.

In a web system, the load balancer is usually the first component of the architecture to interact with incoming user requests. Load balancers are used to assign users' requests to application servers who can compute an adequate response. Two major goals of this component in the web system are A) To minimize the amount of user requests that result in an error and B) To minimize the time it takes for each user to receive a response to their request. These two goals directly affect the QoS of a web application. The load balancer's method for ensuring that these two goals are met is to utilize its load balancing algorithm to adequately distribute the user requests to compute machines. One study conducted by Google suggests that after a request's first 3 initial seconds, the probability that a user will leave the web application is 32% [1]. In the case where the compute servers simply do not have enough resources to handle the incoming load of requests, the application may decide to scale out to accommodate these bursts such that the QoS will not be violated. Cloud providers [2], [3] offer services to automatically scale system resources under developer-defined conditions, such as when the CPU utilization threshold of an application server is met.

While autoscaling techniques prevent QoS violations, increased system-wide resource consumption in turn increases operating costs. To combat unnecessary scale out, certain system components may be configured to operate more efficiently. This tuning can possibly remove the immediate need to spawn additional compute instances. In particular, the intelligent load balancing of user requests can lead to more efficient resource provisioning and usage. As a result, scaling will be done in a more precise manner, as all of the current compute resources are fully saturated upon meeting the scale out threshold.

Current load balancing algorithms are not inherently smart. These algorithms are ignorant to backend metrics or the characteristics of the user requests they are processing. Under some circumstances, a backend server may receive internal processes that it must handle while other servers remain wholly dedicated to accepting incoming requests. In such cases, a request that would be dispatched to such a server in disarray may have been better suited for a server that the

load balancer thinks is under greater stress, when in actuality, is a better candidate for processing the request. On a more granular level, not every request is equal. Each web request may be characterized by features such as its computational workload and memory requirements. Traditional load balancing algorithms are featureless in nature, as they do not consider such circumstances. The nature of current load balancers provoke inefficient cloud systems to commit to the aggressive acquisition of resources.

A major reason that drives this inefficient behavior is the processing of unlikely-to-succeed user requests in the load balancer. With incoming requests, each can be considered to possess a deadline. This deadline can be recognized as a hard constraint, where every request with a corresponding response that has not completed within its deadline is considered a failure i.e. error. Some user requests that are impending to the load balancer may already be considered as futile, as their deadline is unlikely or even impossible to meet. Continuing to process these requests potentially hoards resources for likely-to-succeed requests that are either currently under processing or will be possibly processed in the immediate future.

To make current scaling efforts more efficient, intelligent load balancing decisions can cipher out these unlikely-to-succeed requests, therefore, removing the acquisition of resources that are wasted on them. In turn, these resources can be utilized for requests in which the load balancer is confident will succeed. Being as the application resources will be utilized more efficiently, the system will have less need to scale out, decreasing the overall operation cost.

In this research, an intelligent load balancing algorithm is developed utilizing the ML decisions of a Gradient Boosted Decision Tree model to predict certain metrics of incoming user requests. These metrics are used to better determine which requests can appropriately dispatch to which compute server, such that its deadline will not be missed. In the worst case scenarios in which a request can not be executed on any server within its deadline, the request is dropped, therefore, saving resources for other competing requests. This method of dynamic whitelisting for each request attempts to disallow requests from dispatching to compute servers that can not properly manage them, ensuring that each incoming request will have enough dedicated resources to satisfy its deadline and relieve contention for already-being-executed requests. The key contributions of this paper can be summarized as follows:

- A GBDT model is developed to accurately predict the response time of incoming requests for each backend machine in the web application architecture.
- A GBDT-assisted load balancing algorithm is developed to decrease the need of scale-out and compared against its contemporaries.

II. Background

A. Load Balancing HTTP Requests

In web-related architectures, the system can be broken down into three separate tiers: load balancing, application, and database. An overview of such an architecture can be observed in Fig. 1. The load balancing tier accepts incoming user

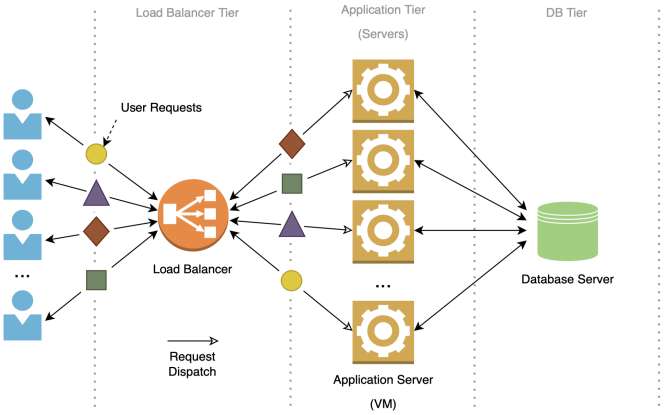

Fig. 1: Architecture of general three-tier web system.

requests. With the load balancing algorithm, an appropriate backend server is determined from a list of possible servers and the request is then dispatched to the application. The application tier's purpose is to satisfy the computational workload that the request brings. The application machines rely on data that is present in the database tier. With incoming requests, the application tier queries the database tier for information used to handle its workload. Once a request has completed its execution in the application tier, the server sends a response to the load balancer to ultimately be returned to the client, completing the transaction.

Web requests are often user-facing, in that there is some deadline the response must meet. The deadline can be considered a concept developed between the inter-client/company relation. One study suggests that as the response time grows, the satisfaction of clients begins to drop linearly [4]. In the circumstance of website hosting, this may result in a loss of traffic and, in turn, company profits. Other situations in which communication is considered mission critical, such as in healthcare environments, slow response times may result in catastrophic failure. The formula for determining the deadline for request instance j of request type i can be viewed in Equation 1, where γ_j is the request instance's dispatch timestamp and ϵ is the Service Level Objective.

$$\delta_j = \gamma_j + \epsilon \tag{1}$$

The dispatch timestamp represents the time in which the load balancer has finished calculating which server to send the request to and actually forwards the message to the application server. The SLO is a static value dependent on the task's type. In this study, we consider that every task type shares the same SLO of 1 second, as this is a fair value to respond to client requests adequately without putting too large a constraint on the system. Completing the execution of the request and forwarding the response to the load balancer within its deadline is the main objective with our architecture. To measure whether or not a request instance has met its deadline on machine m, which is a member of the set M containing all possible machines, its response time e must be captured. Using this value, we can determine whether a

request is deemed successful or not with Equation 2. If a request's response time has surpassed its deadline, then the task instance has failed to meet the QoS standards. The goal with this notion is to maximize the rate of *success/request*.

$$Success_j = \begin{cases} True & e_{j,m} < \delta_j \\ False & e_{j,m} \geq \delta_j \end{cases} \tag{2}$$

Minimizing the amount of requests that result in a deadline miss can be improved through the means of introducing more compute resources. In cloud computing environments, this is usually accomplished by scaling out. Cloud service providers, such as AWS, implement threshold values that developers may define to automatically instantiate more application machines once this value is met. While this protocol may ensure satisfactory client requests, each new application instance introduces additional costs for the developer. Therefore, efficiently scaling out resources is imperative to maximizing system performance while minimizing system costs. This performance-to-cost ratio can be improved by tuning the individual software components of the web service architecture, such as number of processing threads in the application tier or increasing the maximum number of possible UNIX connections in the load balancing tier. Additional to these measures, the actual load balancing algorithm itself can be improved to increase performance metrics.

In the context of this study, incoming web requests that the load balancer receives are treated as tasks. Specifically, each individual web request is an instance of some task type. Identical web requests can be considered to be members of the same task type. Our load balancer is considered to not have any queue by default, but rather receive, process, and dispatch task instances instantaneously. Additionally, the load balancer used during experimentation was designed to be stateless by default.

B. HAProxy Architecture

The load balancing software used during the experimentation studied in this paper is HAProxy [5]. HAProxy is an open-source load balancer meant to be as stateless as possible while maintaining high throughput of messages/second. To facilitate the whitelisting decisions made by our system components, the original code of HAProxy was altered.

HAProxy comes with several load balancing algorithms that are also commonly found in industry, such as roundrobin, least-connection-based, and random. Random, otherwise known as Power of Two [6], randomly pulls two servers from the list of possible backend servers. From these two servers, the algorithm chooses the server with the least current load. This algorithm can further be adjusted to support Power of N where N is any positive, whole number. One can expect that as N approaches the number of actual backend servers, the algorithm's performance will begin to mimic the least-connection-based algorithm. While these algorithms are proven to distribute an incoming workload effectively, there are scenarios in which their performance might prove inadequate. If we consider the list of possible servers an incoming task may be dispatched to as a pool, then each load balancing algorithm only considers servers from this pool in its decision making. This static pooling is prone to ignorance with respect to task characteristics and is not robust to unforeseen fluctuations in server behavior.

C. Related Works

Similar works that involve the scheduling of requests in cloud-related systems study the performance of intelligent decision making load balancers. One such work comes from Kaffes et al. [7]. In this study, the notion of load awareness is used to efficiently prevent unnecessary scale-out and mitigate the cold starts of containerized application environments. This work is similar in that the state of application machines as well as incoming workloads are collected to better facilitate load balancing decisions.

Similarly, another study was conducted that utilizes load awareness scheduling [8]. In an adjacent manner, the characteristics of incoming tasks-to-be-scheduled are treated as a measurable probabilistic value. A mathematical model is developed to utilize this variable for deciding which tasks to "drop", or which tasks should not be considered when scheduling. The goal of this functionality is to increase system *robustness*, or the server's capability in adjusting to unknown workloads and resisting failure.

Wu et al. [9] explores the use of machine learning models in assisting scheduling decisions for cloud applications. Specifically, a GBDT model is used for predicting the time-savings for the merging of similar tasks. Task merging helps reduce the over-consumption of cloud resources as two similar tasks being executed at the same time will lead to unnecessary resource provisioning. The GBDT model's predicted information is used to dictate what tasks are merged to maximize resource efficiency while still meeting deadlines.

Jajoo et al. [10] utilize the varying runtime characteristic of tasks to make smarter scheduling decisions. While the characteristics of a given task may be known prior to the scheduling operation, over time, that same task may begin to exhibit a change in qualities. Using stale task information may lead to inaccurate scheduling decisions. Hence, this paper introduces live task sampling to perpetually maintain up-to-date information about the tasks at hand. This approach ensures that scheduling decisions based upon the characteristics of the tasks at hand are always relevant to the current task behaviors.

The work of this paper is similar to the aforementioned, but differs in that the prediction capabilities of machine learning is introduced to directly guide the load balancing algorithm. The awareness of load and backend states are utilized to bolster load balancing decisions. Alongside these statistics, a machine learning model provides the ability to make outcomes based upon unobservable behaviors that a non-ML approach may not be able to recognize or confidently inference.

III. Methodology

In this study, a three-tier web system was created to replicate a realistic environment that users might find while browsing a website. A dummy blog and ecommerce website was created

Symbol	Metric
i	Task Type
$\bar{e}$	Average Response Time (ART)
σ_{ART}	Standard Deviation of ART
$\bar{s}$	Average Message Size (AMS)
σ_{AMS}	Standard Deviation of AMS

TABLE I: Static Profile Matrix metrics

and propagated with fake pages, items, and posts. This website is used as the application in our architecture. A list of 10 possible task types (web requests) were collected and profiled, obtaining certain metrics such as their average execution time, average message size, and the standard deviation of each amongst instances. The metrics were captured utilizing 30 instances of each task type executed in isolation to ensure no interference from outside resource contention. This profiled information for each task type is stored in the Static Profile Matrix (SPM). The metrics contained in the SPM can be viewed in Table I. Between each task type, there are significant differences in their characteristics. These differences can be exploited in load balancing decisions. However, the behavioral patterns of these differences are not immediately recognized or easily digested. For this reason, a GBDT model is developed to make informative predictions based upon the characteristics of each task type and the current state of the application system.

A. Gradient Boosted Decision Trees

Decision trees are a supervised learning method used to solve regression and classification problems. In each tree, the "leaves" represent some piece-wise function that dictates where a sample may fall given its specific feature characteristics. With gradient boosted decision tree models, an initial tree is formed based upon an average of the given target values during training. Through training iterations, each subsequent tree that is formed utilizes the loss of the prior tree model. In other words, each new tree that is created is learning from the error of the previously created tree, hence the gradient aspect. The purpose of the tree is to ultimately predict some continuous value or discrete classification. In the case of this paper, we utilize the GBDT for regression.

The GBDT model was trained from scratch to predict the response time of every possible task type on every possible application server in the current moment. The hyperparameters for the model were exhaustively explored to determine the configuration presenting the lowest Mean Squared Error (MSE). Once the model was developed, it was trained on a custom dataset.

For training, a total of 100,000 samples were used. The workload used for training the GBDT was synthesized from task types contained in the SPM. To capture the application server state under varying conditions, a range of incoming task rates (instances/second) and task type combinations were used. In particular, the metrics extracted from the server states were the individual CPU utilizations of each server and their respective Expected Response Time (ERT) $\hat{e}$. The ERT is calculated using Equation 3.

$$\hat{e}_m = \sum_{k=1}^{k=i} \sum_{l=1}^{l=j} \bar{e}_{i,j} \tag{3}$$

Essentially, this is a summation of all the Average Response Times of all the task instances on a given server. In a single-threaded machine, we could expect this value to closely resemble an incoming task instance's actual response time, given no interference. To verify the integrity of the dataset and model, k-fold cross validation was used.

Using the variables in the SPM and those gathered from the server, the GBDT model is trained to perform inference and provide the target output of the predicted response time $p_{i,m}$ of a given task type on a given machine at the current time. This information is useful in determining whether or not a task type should be allowed to dispatch to a specific server. Extrapolating these preemptive calculations, a whitelist can be created for each task type with Equation 4.

$$W_i = \left\{ m \in M \mid p_{i,m} < \delta_i \right\} \tag{4}$$

Each task type carries with it a whitelist dictating a list of servers who are predicted to serve the request within its deadline successfully, at the current moment. This form of ML-assisted, dynamic whitelisting allows the load balancing algorithm to make informed decisions on where incoming tasks should be going.

B. Smart HAProxy Architecture

The overall architecture of SmartHAProxy can be viewed in Fig. 2. The original three-tier web architecture still exists, however, our collective SmartHAProxy module sits alongside the original operation. In step 1, data metrics are collected from the application servers. These metrics include the individual CPU utilization of each server as well as task-related events. The Event Monitor observes arriving task instances that have made it to the server but have yet to begin execution. The task type is identified from the instance and its information contained in the SPM is appended to the Dynamic Task Matrix, as shown in step 2. When a task instance has finished execution on the server, the Task Monitor removes the instance from the Dynamic Task Matrix. In essence, the Dynamic Task Matrix encapsulates the load on the application servers in a live manner.

During step 3, the GBDT model observes the metrics captured from the Resource Monitor and the Dynamic Task Matrix state, as well as the historical information from the Static Profile Matrix with step 4. Utilizing this knowledge, the GBDT model then creates a matrix of predictions for each task type and server with step 5. Given the current state of the backend servers and features of the known workload, the model predicts what each task types' response time will be on each given server. Then, this information is passed to the Whitelisting Algorithm. The whitelisting algorithm determines which servers each task type can be dispatched to.

SmartHAProxy's version of the whitelist must be communicated with the Load Balancer in a way that is not obtrusive to either operation. We utilize a shared volume to facilitate

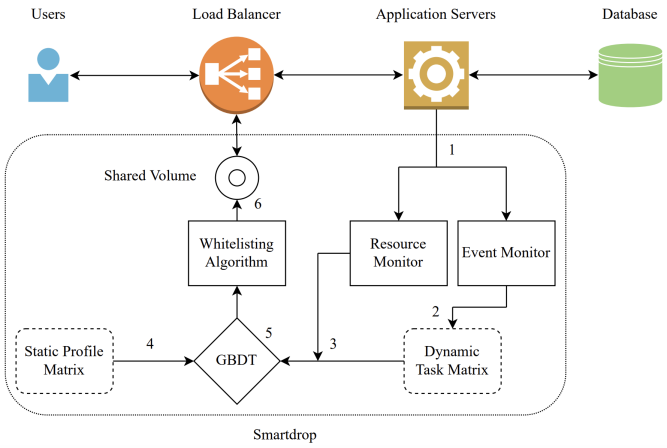

Fig. 2: Architecture of SmartHAProxy. The default three-tier web system lies above. SmartHAProxy sits alongside the original architecture, with its individual modules extrapolated.

the transfer of the whitelist. Whenever the Load Balancer wishes to update its Whitelist, it will message Smartdrop to emplace locks on its computations and log the current state of the whitelist, as highlighted in step 6. Whenever the load balancer is done viewing SmartHAProxy's version of the whitelist, it will notify SmartHAProxy to remove its stall and continue its operation. It is important to consider the overhead that is induced when updating the whitelist. Update it too frequently and significant latency is induced due to network delays. However, updating infrequently leads to the load balancer's local whitelist possessing "stale" data. This stale data will cause the load balancing algorithm to make decisions based upon a system state that existed some time ago, rather than making informed decisions on what is currently happening. Therefore, there is a sweet spot on when the load balancer should decide to update its whitelist to achieve both A) An accurate representation of data and B) Minimal network saturation. After a certain event or period has been reached, the load balancer will request an updated version of the whitelist. In our case, the whitelist will be updated after the load balancer has processed 1,000 requests since the last update or if 10 seconds have passed. The request count threshold value helps the load balancer update its local version of the whitelist frequently enough in the case of increasing traffic, as waiting 10 seconds could result in drastic changes between updates. The periodic update threshold assures the local whitelist is relatively up to date in times of low traffic. This complementary nature between event-based and periodic updating is especially useful in environments that are prone to dynamic traffic rates.

C. Whitelisting Algorithm

The inner workings of the Whitelisting Algorithm can be viewed in Algorithm 1.

In lines 1-7 of Algorithm 1, the Expected Response Time is calculated for each server. The array containing all ERT's is passed to the GBDT model in line 8, alongside the other

Algorithm 1 Whitelisting Algorithm

Require: SPM, DTM, CPU
Ensure: W
1: **for** machine m in DTM **do**
2: $\hat{e}_m \in \hat{E}$
3: **for** task type i in DTM **do**
4: $\bar{e}_i \in \bar{E} \in SPM$
5: $\hat{e}_m$ ++ $\bar{e}_i$
6: **end for**
7: **end for**
8: $P \leftarrow GBDT(\bar{E}, \sigma_{\bar{E}}, \bar{S}, \sigma_{\bar{S}}, \hat{E}, CPU)$
9: **for** task type i in P **do**
10: **for** $p_{i,m}$ in P_i **do**
11: **if** $p_{i,m} < \delta_i$ and $m \notin W_i$ **then**
12: Append W_i with m
13: **else if** m in W_i **then**
14: Remove m from W_i
15: **end if**
16: **end for**
17: **end for**
18: **return** W

metric arrays such as the Average Response Time array ($\bar{E}$), Average Message Size array ($\bar{S}$), and CPU utilization array (CPU). The CPU array contains the individual CPU utilization of each server. The GBDT model returns a matrix where the rows can be considered the task types and the columns the backend servers, where every value at each index represents the Predicted Response Time for a task type on a given server in the current moment. In lines 9-17, the algorithm decides if the PRT violates the task type's deadline. If it does not, the machine corresponding to the PRT value is appended to the task type's whitelist, if it is not already there. If the PRT is greater than the task type's deadline, the corresponding machine is removed from the task type's whitelist, if it is there. The collective whitelist is then returned as the output.

IV. Evaluation

A. Experimental Setup

A view into the hardware and software specifications of our experimental system can be found in Tables II & III, respectively. The web service architecture was established in a mixture of bare-metal, virtual, and containerized machines. The user requests were synthesized using Apache Jmeter [11] from possible requests on the website. The user server sits in its own virtual machine. This removes memory contention from other working machinations of the system and provides a realistic representation of network messaging. The load balancer and application servers each reside in their own container. The application machines are all completely identical in terms of hardware and software. The database is a separate bare-metal machine from the one that the user, load balancer, and application machines reside on. The SmartHAProxy server is also in its own container, providing the ample functionality to plugin to existing systems while minimizing any compromises that should arise.

Tier	Server Count	CPU	Memory
User	1	8 cores @ 2.30 GHz	32 GB
L.B.	1	112 cores @ 2.20 GHz	330 GB
App.	7	16 cores @ 1.80 GHz	32 GB
D.B.	1	48 cores @ 2.87 GHz	132 GB

TABLE II: Hardware Specifications

Purpose	Software
Virtualization	KVM
Containerization	Docker
User Requests	Apache JMeter
Load Balancing	HAProxy
Application	Apache
Web Server	WordPress
Database	MySQL

TABLE III: Software Specifications

During the preliminary tests for our GBDT-assisted load balancing algorithm, we found that the polling interval for updating the load balancer's local whitelist showed the best results when it updated either every 1,000 requests or every 10 seconds. When the request-rate-based updating was too high a value, the load balancer made poor decisions when the incoming traffic was high in volume. Oppositely, unnecessary network delay was induced when the request-rate threshold value was too low. The time-period-based threshold showed similar behavior in terms of its value.

The metrics that were measured in the following tests are the average response time of each request, the percentage of requests that miss their deadline, and the average error rate of each request. These are captured on the application servers' end through the logging mechanism that Apache provides. The average response time is measured by collecting each individual task instance's actual response time and averaging them together. The percentage of deadline misses is measured by checking each individual instance's dispatch timestamp and comparing that to its actual response time. This measurement provides an important insight on to whether the GBDT-assisted load balancing algorithm was correctly allowing a given task type to dispatch to a given server. The average error rate amongst requests is treated as the percentage of total requests that result in some form of an error.

B. Workload

The dummy website created for the experimentation was done through WordPress [12] with various community-sourced plugins. This allowed us to propagate the website with fake pages and content.

Jmeter was used to record the possible HTTP requests that could be sent to the website. These requests are saved as XML files to be used for profiling later. The request features were captured by observing the log data associated with Apache.

The workload used for training the GBDT was synthesized from varying HTTP requests under varying system conditions. The features used as input were recorded using the Event and Resource Monitors. For training, a total of 1,400,000 different samples were used.

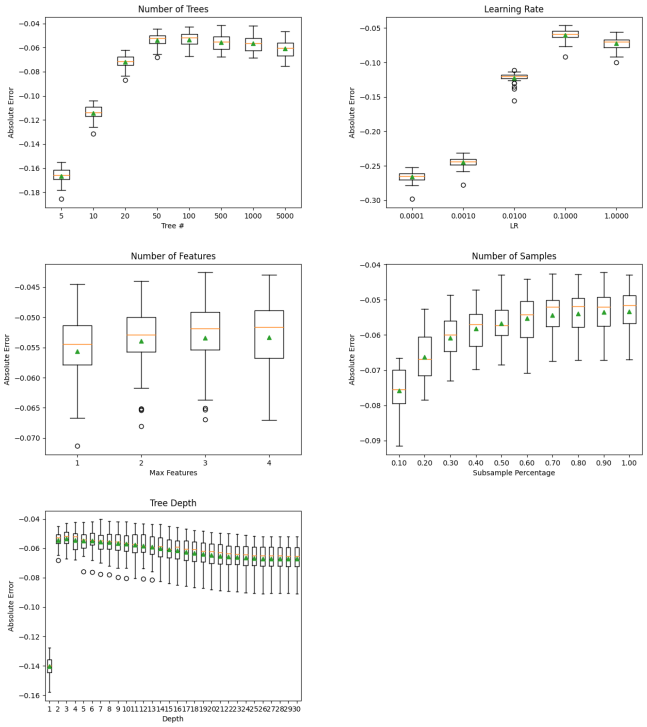

Fig. 3: Hyperparameter performance at varying configurations.

For testing the GBDT-embedded and default load balancers, HTTP requests of a similar manner were used. Each test run contained a varying intensity of workload, with the intensity measured in *requests/second*. Each workload is introduced even over the span of 60 seconds. As an example, the workload intensity of 10 *requests/second* implies a total of 600 requests during the duration of the testing period.

C. Experimental Results

D. Hyperparameter Exploration

During the GBDT model development, the hyperparameters used in tuning were: *number of trees (estimators), learning rate, maximum number of features used per tree, number of dataset subsamples used,* and *maximum depth of each tree*. These results can be viewed in Figs. 3.

We can observe that most of the hyperparameters exhibit some form of diminishing returns. In some cases, such as in the number of trees and tree depth, the model begins to show a decrease in accuracy as the hyperparameter value is increased. The number of trees and learning rate parameters show a large percentage increase between the given points of 5-30 and 0.0001-0.1, respectively, while the other parameters show some form of resilience to changes in value.

Alongside testing each hyperparameter configuration in isolation, a study was conducted with their behaviors documented in tandem. This test was conducted using an exhaustive grid search of all possible configurations given the individual possible values out of the test conducted in Fig. 3. The search space consisted of 12,000 possible configurations, with each model configuration trained on a sample-set size of 10,000.

Hyperparameter	Value
Learning Rate	0.1
Number of Trees	1000
Maximum Features	3
Subsample Size	0.7
Tree Depth	9

TABLE IV: Final hyperparameter values for the GBDT model.

This sample set was pulled randomly from the original dataset of 1,400,000 samples. Ultimately, it was found that the best possible model contained the hyperparameter values found in Table IV. This model was then used to train on the full dataset and then tested against a live workload.

E. Gradient Boosted Load Balancing

Once the selected GBDT model finished training, a test was conducted to examine the GBDT-embedded load balancing system performance against the default, heuristical load balancing approach. The metrics captured that are used to determine performance are the average response time of each request and the error rate percentage of each workload. The results of this test can be observed in Fig. 4.

We can see that as the request rate increases, the average response also increases. This behavior is expected. However, the error rate of each workload does not see a spike until around 75 *requests/second*. When compared to the point at which the requests begin to miss their deadline, this behavior appears to make sense, as all requests up until this point have satisfied their SLO. After requests begin to miss their deadline, errors such as timeouts begin to be raised.

The performance of the GBDT load balancer overall sees a higher average response time, although marginal when compared to the default load balancer. It is assumed that this increase in average response time is due to the necessary protocols taken to collect metrics, query the GBDT model, and communicate with the load balancer, as well as the introduced internal procedures in the load balancer itself such as multithreading lock procedures causing resource contention. While the GBDT load balancer does introduce some overhead, it only begins to cause requests to dissatisfy their SLO at a slightly lower request rate when compared to the default load balancer. In the error rate trend, the GBDT load balancer actually sees lower values in the workload range of 76-81 *requests/second*. However, past this point, the GBDT load balancer responds in high error rates. This suggests that their lies some threshold point for where GBDT-assisted load balancing actually improves system performance, as the ratio of increased response time overhead is marginal compared to the decrease in error rate. The lower error rate percentage in this improved range implies that, while some requests may violate their QoS agreement, currently-being-executed requests on backend servers have a higher chance of success. The reasoning for this is that, for requests that will already violate their SLO, the load balancer will ”drop” them, leaving less resource contention for requests that are already executing.

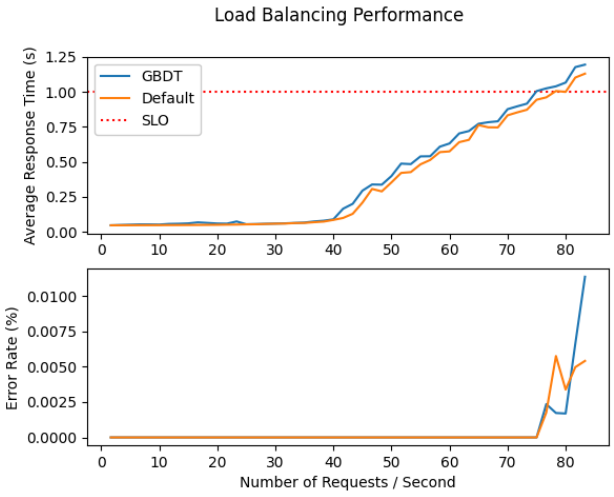

Fig. 4: System performance of each load balancer.

V. Future Works

To increase the performance of the GBDT load balancing system, the average response time increase caused by the aforementioned procedures can be streamlined as to reduce overhead. In particular, the task-related events that are logged an observed by the Event Monitor are done through a shared file resource between it and the load balancer. This process can be assumed to be slow when compared to other procedures such as data messaging through TCP or even HTTP protocols.

Additionally, the whitelist that the load balancer utilizes is updated either when 1) The load balancer has seen 10,000 requests since the last update, or 2) 2 seconds have passed since the last update. The logic behind this is meant to compensate for increased workloads, as using a system state from 2 seconds ago may be considered ”stale” data for large influxes in requests. Ideally, the whitelist should update as frequently as possible. However, this causes issues in the socket communication occurring between the load balancer and whitelisting module. Therefore, a future work for this study could be to determine what proper parameters should be used to ensure that the whitelist is as ”fresh” as possible while mitigating the overhead that is caused by socket communications.

VI. Conclusion

This paper examined the performance of utilizing a GBDT-assisted load balancer for properly distributing HTTP requests in a web architecture. The development of the GBDT model was examined and shown, as well as the affect of each hyperparameter on the model’s training performance. With the results of this study, it is shown that ML has a place in improving system performance in load balancing by providing intelligent decision making over the traditional heuristical approach. As the realm of web systems begins to expand beyond just simple websites, the need for smarter load balancing systems accompanies the requirement for fast response times and low error rates.

REFERENCES

[1] "Google/soasta research, 2017," https://www.thinkwithgoogle.com/marketing-strategies/app-and-mobile/mobile-page-speed-new-industry-benchmarks/, accessed: 2022-08-03.

[2] "Aws auto scaling," https://aws.amazon.com/autoscaling/, accessed: 2022-08-03.

[3] "Azure autoscale," https://azure.microsoft.com/en-us/products/virtual-machines/autoscale/, accessed: 2022-08-03.

[4] J. A. Hoxmeier and C. DiCesare, "System response time and user satisfaction: An experimental study of browser-based applications," 2000.

[5] "Haproxy," https://github.com/haproxy/haproxy, accessed: 2022-08-03.

[6] A. W. Richa, M. Mitzenmacher, and R. Sitaraman, "The power of two random choices: A survey of techniques and results," *Combinatorial Optimization*, vol. 9, pp. 255–304, 2001.

[7] K. Kaffes, N. J. Yadwadkar, and C. Kozyrakis, "Practical scheduling for real-world serverless computing," *arXiv preprint arXiv:2111.07226*, 2021.

[8] A. Mokhtari, C. Denninnart, and M. A. Salehi, "Autonomous task dropping mechanism to achieve robustness in heterogeneous computing systems," in *2020 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, 2020, pp. 17–26.

[9] S. Wu, C. Denninnart, X. Li, Y. Wang, and M. A. Salehi, "Descriptive and predictive analysis of aggregating functions in serverless clouds: The case of video streaming," in *2020 IEEE 22nd International Conference on High Performance Computing and Communications; IEEE 18th International Conference on Smart City; IEEE 6th International Conference on Data Science and Systems (HPCC/SmartCity/DSS)*. IEEE, 2020, pp. 19–26.

[10] A. Jajoo, Y. C. Hu, X. Lin, and N. Deng, "A case for task sampling based learning for cluster job scheduling," in *19th USENIX Symposium on Networked Systems Design and Implementation (NSDI 22)*, 2022, pp. 19–33.

[11] "Apache jmeter," https://github.com/apache/jmeter, accessed: 2022-08-03.

[12] "Wordpress," https://github.com/WordPress/WordPress, accessed: 2022-08-03.