ReTail: Opting for Learning Simplicity to Enable QoS-Aware Power Management in the Cloud

Shuang Chen, Angela Jin, Christina Delimitrou, José F. Martínez
Computer Systems Laboratory
Cornell University
Ithaca, NY, USA
{sc2682,aj445,delimitrou,martinez}@cornell.edu

***Abstract*—Many cloud services have Quality-of-Service (QoS) requirements; most requests have to to complete within a given latency constraint. Recently, researchers have begun to investigate whether it is possible to meet QoS while attempting to save power on a per-request basis. Existing work shows that one can indeed hand-tune a request latency predictor offline for a particular cloud application, and consult it at runtime to modulate CPU voltage and frequency, resulting in substantial power savings.**

In this paper, we propose *ReTail*, an automated and general solution for request-level power management of latency-critical services with QoS constraints. We present a systematic process to select the features of any given application that best correlate with its request latency. ReTail uses these features to predict latency, and adjust CPU's power consumption. ReTail's predictor is trained fully at runtime. We show that unlike previous findings, simple techniques perform better than complex machine learning models, when using the right input features. For a web search engine, ReTail outperforms prior mechanisms based on complex hand-tuned predictors for that application domain. Furthermore, ReTail's systematic approach also yields superior power savings across a diverse set of cloud applications.

I. Introduction

Warehouse-scale datacenters host an increasing number of latency-critical (LC) interactive services, such as websearch and social networks [12, 35, 39]. These services operate under strict Quality-of-Service (QoS) constraints in terms of *tail latency*—meaning that, for example, the 99^{th} percentile of requests need to complete within a latency threshold to guarantee predictable performance and good user experience.

In addition to meeting tail latency requirements, energy efficiency is also important for datacenters, which incur billion-dollar energy bills each year [23, 39, 49]. To account in part for load surges and unexpected spikes, servers running LC applications are often provisioned for the worst case. This means that under normal conditions, they are severely underutilized [12, 19, 35, 46]. The same applies to power management: LC services usually operate at high frequency to avoid latency spikes [34]. This is neither economical nor necessary, given the cloud's well-documented resource underutilization [12, 19, 20, 35], calling for more fine-grained power management that accurately captures the real-time performance needs of interactive, LC services.

Prior work has explored dynamic voltage/frequency scaling (DVFS) for LC services [23, 24, 29, 34, 39]. Pegasus [34] dynamically adjusts the frequency for an LC application by monitoring its load and latency slack. Unfortunately, a key characteristic of LC services is their large variation in request latency: a few requests are orders of magnitude slower and end up defining the application's tail latency [17, 33]. Thus, power management at application granularity misses the opportunity to differentiate frequency at the level of individual requests. There has been work that explores fine-grained request-level power management. Some proposals use heuristics [23, 24] to classify requests into short and long. Others use statistical models [29] to estimate worst-case latency, which is often too conservative. The most recent related work, Gemini [51], uses application-specific neural networks to predict request latency for websearch, without generalizing to other LC services.

In this paper, we present *ReTail*, the first automated framework using *practical, simple linear techniques* to enable QoS-aware power management of LC cloud services with request-level latency prediction. ReTail shows that simple techniques using appropriate input features are much more effective that complex ML models treated as black boxes. We motivate the design of ReTail through a characterization of seven diverse LC applications, showcasing the opportunity to accurately infer their request latencies from selected request/application features. ReTail has three main components:

1) **Feature selection** automatically identifies the features that most closely correlate with request latency for a given service. ReTail expands the feature space by including features not immediately available at request arrival.
2) **Latency prediction** uses linear regression to predict the request-level latency based on the selected features. Compared to more complicated models, such as neural networks, linear regression is accurate, simple, and general. It achieves similar accuracy with significantly lower overheads, and does not require manual tuning per application.
3) **Runtime power management** uses the latency predictor's output to estimate the end-to-end latency under different frequencies, and to determine the minimum frequency for each request to meet the end-to-end QoS.

We evaluate ReTail on a high-end server platform across various representative LC applications. We compare it with two state-of-the-art request-level power management schemes, Rubik [29] and Gemini [51], and show that ReTail reduces power consumption more than either mechanism, by up to 36.2% and 35.6% (average 11.5% and 8.9%), respectively, and without dropping any requests. Additionally, ReTail is able to

TABLE I: Qualitative comparison of ReTail vs. two closely related proposals for power management of latency-critical applications.

	Rubik	Gemini	ReTail
Method	Statistical model	Neural net	Linear regression
Feature space	N/A	Request	Request&Application
Feature selection	N/A	Hand-picked	Systematic
Request-accurate	✗	✓	✓
General applications	✓	✗	✓
Training overhead	Low	High	Low
Inference overhead	Low	High	Low
QoS guarantee	✓	✗	✓
No dropped requests	✓	✗	✓
Adapt to model drift	✗	✗	✓

dynamically adjust to system or application interference by retraining the latency predictor online.

II. Related Work

Most proposals towards improving power efficiency [15, 26, 43] focus on throughput-oriented batch jobs. Recent work that explores DVFS for LC services with latency/QoS constraints can be broadly classified into two categories:

Coarse-grained power management like Pegasus [34] dynamically adjusts frequency for the entire LC application. It addresses long-term latency variations under load fluctuations, but leaves power savings on the table by not differentiating individual requests.

Fine-grained power management outperforms coarse-grained management by differentiating at request granularity, generally boosting long and slowing down short requests, while trying to meet QoS. There are two main methods:

- *Classification-based methods* classify requests into short and long using various metrics, and boost all requests in the long category. EETL [23] tracks the progress of each request; those that exceed a predetermined execution threshold are flagged as long requests, and EETL boosts the frequency at that point. However, by the time a request reaches the progress threshold, it may be too late to prevent tail latency degradation. Adrenaline [24] introduces the concept of feature-driven request classification. It studies two LC applications, web search and key-value stores, and identifies, *by human inspection*, request features in each that can be used to predict incoming requests as being either "short" or "long." Adrenaline uses this classification to boost long requests from the start. Unfortunately, Adrenaline does not easily generalize to other LC services, whose features will be different. Additionally, a downside of request classification is that it cannot rank requests within each category, and therefore, is not fine-grained enough to accurately pinpoint requests at the tail; instead, an entire class of requests are boosted when only the longest ones needed to. To address this issue, ReTail instead follows the latency-based approach described next.
- *Latency-based methods* take a step further over classification, by using latency prediction to guide power management. Recent work [29, 51] has shown great improvement over classification-based methods. Rubik [29] estimates a latency distribution for each request based on the current queue length and a request service time distribution (the latter profiled offline). It then uses the tail of the estimated distribution as the predicted latency. This is usually too conservative, especially when the resulting latency distribution has a long tail. Gemini [51], most related to our work, predicts per-request latency using a neural network. Gemini is designed specifically for web search engines, and both the features and its prediction model are hand-picked for the specific service, similar to prior work [28, 31, 36] that also predicts request latency for web search engines. It also does not provide a QoS guarantee, as QoS violations still occur in some cases, e.g., upon wrong latency predictions/high loads.

Table I shows the main differences between ReTail and two latency-based methods. Compared with Rubik and Gemini, ReTail targets general LC applications, and adopts much simpler linear regression for latency prediction, automatically selecting the request features that impact latency most. ReTail dynamically detects and adjusts to model drift, and does not need to drop requests. We compare with Rubik and Gemini in Sections V-A and VII.

III. The Latency Impact of Request Features

We now characterize seven diverse LC applications (Table II) from Tailbench [30], to discover any underlying latency patterns, and explore opportunities for latency prediction. These applications are widely used in both academia and industry, giving us a realistic pool of LC workloads to study.

A. Service Time Distribution

Request latency (sojourn time) is the sum of service time (the time to process a request) and queueing delay (the waiting time before a request starts execution). Fig. 1 shows that service time of a 20-threaded ImgDNN remains constant, while sojourn time increases with request-per-second (RPS) due to the increase in queuing delay. Throughout all Tailbench applications and many other LC applications [14], no batching is adopted during request processing to preserve low request latency, i.e., a single request is processed at a time, and once started, it runs to completion.

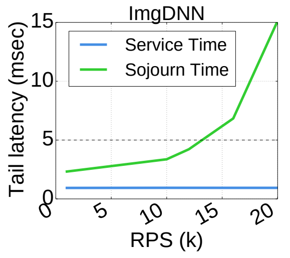


Fig. 1: Tail latency with RPS.

In this section, we focus on characterizing *service time*. Queuing delay is incorporated in Section VI to predict sojourn time. Fig. 2 plots each application's service time cumulative distribution function (CDF); median and tail latencies are marked with green stars and red circles, respectively, and the median-to-tail ratios are recorded in Table II. The x-axis distance between the two markers visually indicates the service time variation across all requests. Among the seven applications, Masstree and ImgDNN have little or no variation in service time: the median is less than 20% of the tail, i.e., most requests are serviced in roughly the same time.

TABLE II: Latency-critical applications.

Application	Masstree	ImgDNN	Sphinx	Xapian	Moses	Shore	Silo
Domain	Key-value store	Image recognition	Speech recognition	Web search	Real-time translation	Database (disk/SSD)	Database (in-memory)
Dataset	One million <key,value> pairs	MNIST [21]	CMU AN4 [11]	English Wikipedia	Spanish articles [6]	TPC-C [16], 1 warehouse	
QoS Target	1ms	5ms	4s	8ms	120ms	5ms	1ms
Median:Tail Ratio	0.84	0.81	0.36	0.27	0.26	0.25	0.19
Request	90% <GET, key> 10% <PUT, key, value>	An image with a handwritten digit	Path to an audio file	A single-word term	A Spanish phrase to be translated into English	47% PAYMENT 45% NEW_ORDER 4% ORDER_STATUS 4% STOCK_LEVEL	
Classification	Little or no variation	Little or no variation	Predicted by request features	Predicted by application features	Predicted by request features	Predicted by request and application features	
Feature(s)	N.A.	N.A.	Audio file size	Document count	Word count	Request type, Item count, Rollback	

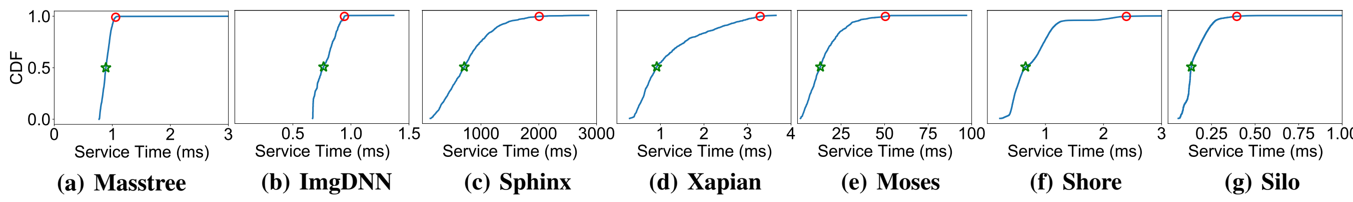


Fig. 2: CDF of service time. Median and 99th-percentile latency are marked by green stars and red circles.

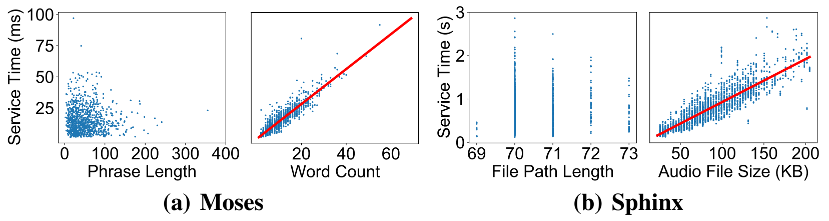


Fig. 3: Correlation between various interpretations of request length and request service time. Each blue dot represents one request sample.

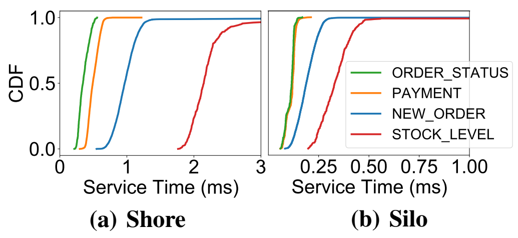


Fig. 4: CDF of service time of each request type.

B. Correlation with Request Features

To understand the factors influencing service time variation in the remaining five applications, we start by exploring request features, namely request length and request type.

1) Request length: Requests of Sphinx and Moses have different lengths. Since request length can have multiple interpretations, it is critical to identify all of them. Fig. 3 shows the scatter plots between various definitions of request length and service time.

For Moses, whose requests are phrases, request length can be defined as phrase length (number of characters in the phrase), or as word count (number of words in the phrase). Despite this seemingly subtle detail, Fig. 3a demonstrates that only word count correlates with service time. Intuitively, a longer word does not take longer to translate than a shorter word, but a phrase with many words indicates a complex structure, and will likely induce a longer translation time.

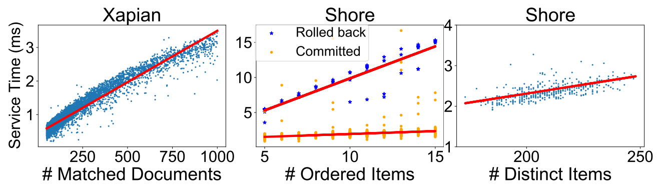
Similarly, for Sphinx, whose requests are paths to audio files, request size can be naïvely defined as path length, or, more meaningfully, as the audio file size. Fig. 3b shows that only file size correlates with service time.

2) Request type: Shore and Silo both use the same TPC-C benchmark, so they have the same types of requests. Fig. 4 shows their per-type service time CDF graphs. The curves of ORDER_STATUS and PAYMENT both exhibit a nearly vertical rise from 0.0 to 1.0, i.e., all requests of the same type have similar service times. NEW_ORDER and STOCK_LEVEL requests have more service time variation, and we further investigate below.

C. Correlation with Application Features

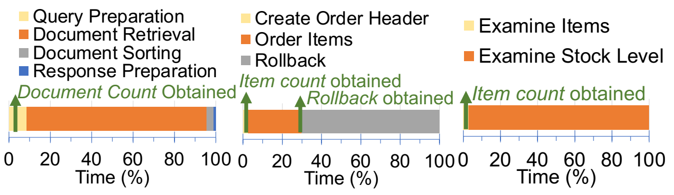
Xapian and two types of requests in Shore and Silo require further investigation into their service time variations. Due to the lack of useful features in their request packets, we look into intermediate variables in each application, i.e., application features.

1) Xapian: Fig. 6a shows four major steps during Xapian's request processing. In the first step, we find an intermediate variable titled *term frequency*, which correlates strongly with service time, as shown in Fig. 5a. *Term frequency* represents the number of documents matched to a searched term, and the correlation can be interpreted intuitively: the more matched



(a) Xapian **(b) NEW_ORDER** **(c) STOCK_LEVEL**

Fig. 5: Correlation between application features and service time.



(a) Xapian **(b) NEW_ORDER** **(c) STOCK_LEVEL**

Fig. 6: Lifetime of a request.

documents, the longer it takes to retrieve and sort the documents (step 2&3 in Fig. 6a).

2) NEW_ORDER in Shore and Silo: a new order is entered into the database, by first creating an order header and then inserting a new row to the ORDER_LINE table for each ordered item [16] (Fig. 6b). Upon user data entry errors, the transaction is rolled back. Therefore, request service time depends on (a) *if the transaction is rolled back*, which incurs additional operations to remove previously inserted rows, and (b) *the number of ordered items*, since both ordering items and rollback primarily consist of a for-loop with #items iterations. Fig. 5b shows that Shore's processing times of these two steps each increases with the item count, but at different rates. Since Silo and Shore have similar application logic (but different underlying implementations to store data), the features that correlate with service time are the same.

3) STOCK_LEVEL in Shore and Silo: A STOCK_LEVEL request examines the stock level of all the items on the last 20 orders; it first examines all items on the last 20 orders, and then examine each distinct item's stock level [16] (Fig. 6c). Fig. 5c shows that processing time increases with the number of distinct items, as it takes longer to examine the stock level of all the items.

4) Timeliness of application features: Unlike request features, application features are obtained during request processing, which means that latency cannot be inferred before request processing. Luckily, application features in Xapian, Silo and Shore are all obtained early during request processing, as shown in Fig. 6. Timeliness ensures the feasibility of using application features in service time prediction (further discussed in Section IV-B).

D. Summary

We successfully identified the cause of latency variation for every characterized application. Based on the selected

TABLE III: Symbols in ReTail feature selection.

Symbol	Definition
M	Number of candidate features.
N	Number of request samples.
$Start_i, End_i$	Start and end time of request i's processing, $1 \leq i \leq N$
$Service_i$	Service time of request i, i.e., $End_i - Start_i$.
$Feature_{i,j}$	Feature j's value of request i, $1 \leq j \leq M$
$FArrival_{i,j}$	Feature j's arrival time of request i.
$Lateness_j$	The fraction of service time to obtain feature j's value, defined as $\frac{\sum_{i=1}^{N}(FArrival_{i,j} - Start_i)/Service_i}{N}$

feature(s), we can categorize applications into four categories:

- **Little or no variation (ImgDNN, Masstree)):** All requests are serviced in the same amount of time.
- **Predicted by request features (Moses, Sphinx)**: Some interpretation of request feature (e.g., request size/type) strongly correlates with service time.
- **Predicted by application features (Xapian)**: No meaningful request feature is known before execution. Instead, an application feature, ideally obtained early in the request execution, can explain the variation in service times.
- **Combinational (Shore, Silo)**: Request and application features are both needed to jointly explain the variation.

All correlations have very intuitive explanations. The identified features also generalize across all applications with the same functionality. For instance, Shore and Silo, which are both database applications, share the same features. *Document count* and *word count* are likely to correlate with service time in other websearch and translation services, respectively.

IV. Feature Selection

We have shown that LC applications generally have some features that can explain their service time variation. We now introduce ReTail's *feature selection* that automatically selects critical features for a general LC application.

A. Input of Feature Selection

The input to ReTail's feature selection for application A consists of M candidate features and N request samples (see Table III). For each request sample i, $Start_i$ and End_i are required to obtain request service time $Service_i$. For each feature j of request i, the feature value $Feature_{i,j}$ and the time to obtain the feature $FArrival_{i,j}$ are required.

1) Input Features: The cloud user of A is responsible for providing a *full list* of candidate features, including request and/or application features. Not to be confused with handpicking features, this list is unfiltered and can be as long as needed; it is ReTail's job to *automatically* pick the final features that most closely correlate with latency.

2) Input Requests: Once the application and the list of candidate features are submitted, the cloud provider collects all the request samples using live traffic. The application is deployed at a fixed frequency in isolation, to rule out the impacts of runtime frequency fluctuations and resource contention on per-request service time variation. N should be large enough to cover a representative subset of all kinds of requests. For instance, our characterization of Tailbench applications uses load generators that randomly generate requests with various request lengths/types, and we find $N = 1000$ to be sufficient.

B. Feature Selection

ReTail's feature selection consists of the following steps to systematically select one or more distinguishing features that correlate with request service time for an LC application:

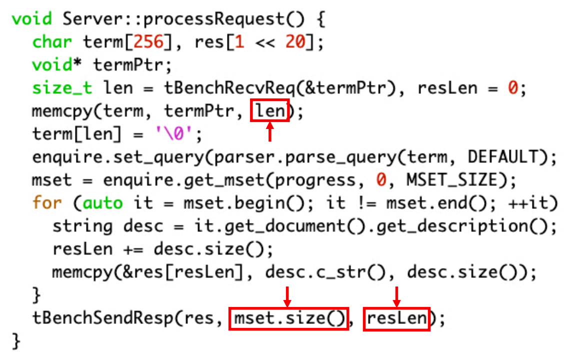
Step 1: Calculate each candidate feature's *lateness* $\in [0, 1]$, defined in Table III as the fraction of service time that has elapsed when the feature value is obtained. Request features have $lateness = 0$. We rule out all features with $lateness$ greater than 0.5, since these features' values cannot be obtained until after half of the request processing time has elapsed, which is too late for benefiting from frequency adjustment. For other purposes that may benefit from such features, this threshold can be adjusted.

Step 2: Calculate the *correlation degree* (CD) of each candidate feature, and sort all the features in decreasing order of CD. Features are either numerical or categorical.

- For numerical features (e.g., request size), correlation degree is defined as $|\rho|$, the absolute value of the Pearson correlation coefficient (ρ) [10, 48] between the feature and service time. $|\rho| \in [0, 1]$; $|\rho|$ closer to 1/0 indicates strong/no linear correlation. The use of the Pearson coefficient over other correlation coefficients is guided by the observation that correlation relationships are usually linear (Fig. 3 and 5).
- For categorical features (e.g., request type), correlation degree is defined as η^2, the square of the correlation ratio [3]. $\eta^2 \in [0, 1]$; η^2 closer to 1 indicates less variation within each category. $\eta^2 = |\rho|$ when the relationship is linear.

Step 3: We follow the *forward stepwise feature selection algorithm* [25] to select features that correlate the strongest with service time. We first select the top feature with the highest CD, and then repeatedly add one more feature at a time until CD cannot be improved further. Note that we do not simply select the top few features. Redundancy in features is avoided by only selecting a feature only if it improves the overall CD; CD will remain unchanged if multiple features capture the same information about a workload. We define CD of multiple features as follows:

- Multiple categorical features are merged into a single categorical feature with $\prod_{i=1}^{n} a_i$ categories, given n categorical features that each have a_i categories. Without numerical features, CD is defined as η^2 using the merged feature.
- *Multiple correlation coefficient* [9], an extended version of the (Pearson) correlation coefficient, is used to calculate correlation degree for multiple numerical features in the absence of categorical features.
- For combinations of categorical and numerical features, correlation degree is defined as the average correlation degree using numerical feature(s) across all the categories.
- When considering a new feature f_{new} to the current pool of features $f_{1..k}$, we say correlation degree is improved if $CD(f_{1..k}, f_{new}) > CD(f_{new})$ if f_{new} is the first numerical feature, or $CD(f_{1..k}, f_{new}) > CD(f_{1..k})$ if otherwise.



```
void Server::processRequest() {
  char term[256], res[1 << 20];
  void* termPtr;
  size_t len = tBenchRecvReq(&termPtr), resLen = 0;
  memcpy(term, termPtr, len);
  term[len] = '\0';
  enquire.set_query(parser.parse_query(term, DEFAULT));
  mset = enquire.get_mset(progress, 0, MSET_SIZE);
  for (auto it = mset.begin(); it != mset.end(); ++it) {
    string desc = it.get_document().get_description();
    resLen += desc.size();
    memcpy(&res[resLen], desc.c_str(), desc.size());
  }
  tBenchSendResp(res, mset.size(), resLen);
}
```

Fig. 7: Identified application features (red arrows) in function `Server::processRequest`

C. Discussion

How to identify all candidate features? If the cloud users are also the application developers, they can leverage their knowledge of the application to provide a list of features (note that developers do not need to know which features will be critical, just which features *exist* in requests and applications). Even if developers are not confident in the feature list, candidate features can still be easily identified, leveraging our successful experiences with selecting features for Tailbench as outsiders:

- Our characterization shows that more than half of the services do not require any application feature. Simply providing request type and request size is sufficient.
- When application features are necessary, to avoid hand-picking them, the simplest and most effective approach is to leverage the tracing statements embedded by the application developers. It is typical in software development to insert event logging messages throughout a program, for debugging and monitoring purposes [44, 50]. All Tailbench applications have embedded /logging that records important variables. These variables form the list of candidate features.
- Alternatively, profiling tools (e.g., `perf record --call-graph` and `perf report`) can be used to collect application stack traces and identify intermediate variables created or used in the traced functions. For instance, Fig. 7 shows the application features identified for Xapian (marked by red arrows) in the source code of `Server::processRequest`, the top function in Xapian's stack trace.

Note that once the final features are selected, application's source codes have to be modified to explicitly expose the selected features (more details in Section VI-A).

What if the user is not willing to share information with the cloud operator? Since ReTail requires information about application features, concerns about application privacy are valid, especially when running on a public cloud. Fortunately, ReTail need not know what the features are or mean; it only needs labeled values. Both labels and values can be obfuscated by the application, as long as obfuscated values remain linear to service time. In fact, an increasing amount of computation can be performed over encrypted data [41, 42, 47], and ReTail can make use of these mechanisms to preserve the application's privacy. Furthermore, the feature selection itself can be offloaded to the client's side, with the user only providing ReTail with the list of (obfuscated) features to use. In any case, we note that an increasing number of applications running in public clouds are hosted in a way where the cloud provider has access to the application's source code. In serverless and Function-as-a-Service (FaaS) frameworks, for example, users upload their code to the cloud, and requests are executed under an event-driven model, on short-lived instances. In this case, the cloud provider already has access to the application logic and incoming requests, therefore ReTail does not leak any additional information about the workload.

What if there are interactions between features? Redundant features that strongly correlate with other features are explicitly handled as described in Section IV-B. For other types of cross correlation between features, although we did not observe any, if it occurs, it can be supported by including pairs/groups of features in the first two steps of feature selection. While this slightly increases the feature space, in practice, usually a small number of features (1-2 in our case) is enough.

Note that ReTail is designed based on observations from Tailbench. It is possible that applications do not have features that correlate with request service time, and there might be complex feature interactions, such as XOR relationship, both of which ReTail currently does not consider.

V. Latency Prediction

We now introduce ReTail's *latency prediction*, which infers, for a given application, a request's service time under a certain frequency. We first introduce the linear regression prediction model, then describe the training dataset, and finally, elaborate on our training, retraining, and inference processes.

A. Prediction Model

Suppose an application has n categorical features (each with a_i categories) and m numerical features, and the system has k frequency settings. Our goal is to build a prediction function that takes in, as input, the $n+m$ feature values of a request and a target frequency, and outputs the predicted service time of this request under the target frequency. Specifically, the prediction function is composed of $k \times \sum_{i=1}^{n} a_i$ separate prediction functions, each handling prediction for one combination of categorical values under each frequency setting. This is because prediction functions may vary significantly across different categories (e.g., NEW_ORDER and STOCK_LEVEL requests in Shore cannot share the same prediction function).

We build a separate prediction model under each available frequency setting since latency is not necessarily linear/proportional to frequency. Rubik and Gemini both assume a proportional relationship between frequency and latency, and simply scale the latency prediction at a certain frequency up and down for prediction under other frequencies. We find that this does not hold for typical LC applications, especially non-compute-intensive ones.

TABLE IV: Quantitative comparison of three prediction models: linear regression, Gemini's neural network model (NN-G), and an optimized hand-tuned NN model (NN-T) which requires careful hand-tuning for each application. Each NN model is described using a tuple of of (#layer, #neuron/layer, #epoch, batch size).

	Model Info	Overhead		Accuracy	
		Training	Inference	R^2	RMSE/QoS
Xapian	Linear Regression	$0.003s$	$5\mu s$	0.959	4.18%
	NN-G: (5, 128, 15, 32)	$9.71s$	$363\mu s$	0.973	3.38%
	NN-T: (1, 16, 5, 32)	$0.98s$	$107\mu s$	0.974	3.30%
Moses	Linear Regression	$0.003s$	$5\mu s$	0.854	3.02%
	NN-G: (5, 128, 500, 32)	$85.10s$	$514\mu s$	0.833	3.22%
	NN-T: (1, 4, 400, 1024)	$0.74s$	$258\mu s$	0.854	3.01%
Sphinx	Linear Regression	$0.003s$	$5\mu s$	0.746	5.45%
	NN-G: (5, 128, 1000, 32)	$36.15s$	$344\mu s$	0.747	5.43%
	NN-T: (3, 128, 700, 32)	$15.39s$	$300\mu s$	0.747	5.43%

For applications with only categorical features, we simply predict latency under a specific combination of categories and a certain frequency using the average service time of all the requests under the category combination and the target frequency. Applications with little-to-no variation can be treated as applications with a single category. In the rest of this section, we focus on building the separate prediction function for applications with numerical features.

B. Prediction Model of Numerical Features

We use the following metrics to evaluate a model:

- $\boldsymbol{R^2}$ [2]: A goodness-of-fit measure for predictions, normally ranges from 0 to 1. Higher is better.
- **Root-mean-squared error (RMSE)** [27]: An aggregated measure of the differences between predicted and observed values. Smaller is better. We evaluate **RMSE**, by dividing RMSE by the application's QoS target, to show the importance of RMSE. For instance, increasing RMSE from 0.1ms to 1ms may seem significant degradation, but if the application's QoS is 1s, RMSE of 1ms is still negligible.
- **Training overhead**: The time it takes to train the model, which is less important if the model is trained offline. However, LC applications usually experience inevitable system interference and/or dynamic resource allocations; both affect service time and model accuracy. The prediction model only captures application-level sources of latency variation, and has to be updated or completely retrained online upon such model drifts. Models with high training time are unable to quickly adapt to such environment changes at runtime, and result in longstanding QoS violations.
- **Inference overhead:** Time to get a latency prediction. Since the inference will be triggered (multiple times, as introduced in Section VI-B) for every request, keeping inference overhead negligible to the request latency is critical.

We start from linear regression (LR), one of the simplest approaches to model relationship between variables, fitted using the ordinary least squares method, which minimizes the sum of squared residuals [22, 27]. Specifically, under a specific combination of categories and a certain frequency, the prediction function $L(f_1, ...f_m)$ equals to $\sum_{i=1}^{m} a_i * f_i + b$, where $f_1 ... f_m$ are the m numerical features, and $a_1 ... a_m$ and b are constants. As shown in Table IV, LR achieves pretty high accuracy: R^2 is close to 1, and the prediction error (RMSE) is less than 6%. We find various advantages of the simple LR:

1) Section III shows that the relationship between numerical features and service time is rather simple. The good alignment between each red solid line (LR prediction) and the scatter plot trend in Fig. 3 and 5 shows that the simple LR clearly captures each application's service times.
2) LR incurs minimal training and inference overheads. Training takes only $3ms$ per request on average, making online retraining very affordable. Inference takes only $5\mu s$ each time, negligible to most if not all LC applications.
3) The process of building a LR model is generalizable across applications. Once features are selected, each feature corresponds to one variable in the regression model (e.g., $f_1 ... f_m$). Coefficients ($a_1 ... a_m$ and b) are automatically calculated through training.
4) LR is easily explainable, which, as opposed to the ML black-box model, can lend itself to insights for software optimizations. For instance, Xapian's service time increases almost linearly with the number of matched documents. Given this, we could split a large request into two, each with a smaller *doc count*, to reduce its service time.

Despite these advantages, more complicated models may further improve accuracy. Taking Xapian as an example, the scatter plot in Fig. 5a shows a slightly concave trend.We further investigate Xapian's service time decomposition in Fig. 6a. Suppose *document count* is d. Then, the time complexity of query and response preparation are $O(1)$ each, of document retrieval is $O(d)$, and of document sorting is $O(dlogd)$. We attribute the curved scatter plot pattern to the document sorting time complexity. Therefore, we also explore neural networks (NN), known for their ability to identify almost any underlying relationship between variables. LR and NN fall at the two ends on the spectrum of prediction models: the former being simple but potentially less accurate, and the latter being complicated but more powerful. Comparing these models helps us understand the tradeoff between accuracy and overhead when predicting service times.

NN-based request latency prediction was proposed in very recent work, Gemini [51]. Gemini's NN model has 5 hidden layers and 128 neurons/layer. It uses the ReLU activation function and MSE (mean-square error) loss function. We implemented NN in PyTorch [40] and reproduced Gemini's NN model (shown as **NN-G** in Table IV), using the same input features as those used for LR. We observe some accuracy improvement for Xapian, visualized in Fig. 8: NN-G captures the concave nature between the number of matched documents and service time. However, the zigzag pattern around $\#\ matched\ documents = 450$ indicates that the model is slightly overfitting. Because Gemini is designed for *Apache Lucene Search* [1], the exact NN model proposed cannot be generalized to other LC applications. We also find negligible accuracy improvement for Moses and Sphinx, but more than $3000\times$ increase in training time and more than $60\times$ increase in inference time.

Therefore, we manually tune the NN model for each LC application to find the configuration with the least overhead without losing accuracy. We first choose a large epoch at which accuracy has converged. Then, we successively tune batch size, the number of hidden layers, the number of neurons per layer, and the number of epochs in order, to reduce the training overhead while maintaining accuracy. Table IV shows the results of our hand-tuned optimized NN model, shown as **NN-T**. Sometimes, accuracy even increases because smaller NN structures reduce the likelihood of overfitting, as demonstrated by Xapian and Moses. Fig. 8b also shows that the line is smoother under NN-T. Training and inference overheads are both significantly reduced compared to NN-G, but are still much higher than LR. Note that NN-T does not offer generalization over NN-G; neural networks always require careful tuning of a number of parameters to find the best configuration for each application.

To conclude, LR is the best model for our problem setting. NN improves accuracy slightly, but at the expense of orders of magnitude higher training&inference time.

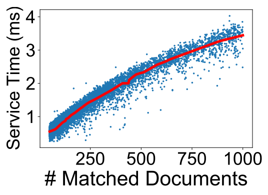
C. Training Dataset

The predictor uses a training dataset that includes the following information for each training sample:

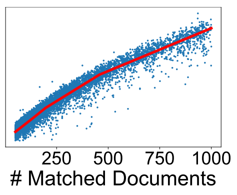
- **Frequency**: The frequency the request ran at.
- **Queueing delay**: The time elapsed after a request is generated and before it starts processing. We pass the request generation timestamp in wall time (t_1) to the server through the request packet, and take another timestamp (t_2) when the packet is received by the application. Queueing delay is calculated as $t_2 - t_1$.
- **Feature values**: The values used for service time prediction. Unlike the input for ReTail's feature selection, only values of selected features are needed, imposing negligible overhead to the application to provide these information.
- **Sojourn time**: The end-to-end request latency. We take a timestamp (t_3) when the client has received the response packet. $t_3 - t_1$ and $t_3 - t_2$ produces the sojourn time and the service time, respectively.

Data collection and training/retraining both happen online, using live data. When a new application comes to the system, we set the system frequency to the lowest setting and progressively increase the frequency until it reaches the maximum configuration. We collect information for 1000 requests at each frequency. As shown in Fig. 9, we did a sensitivity study on the training dataset size, and found that 1000 samples[1] are sufficient for the model (i.e., R^2) to converge for all applications. Depending on the application's real-time RPS, data collection takes less than one second most of the time (when RPS $\geq$1000), but may take up to tens of seconds, e.g., for Sphinx, whose request service time is up to $3s$ and whose

[1]Even 100 samples are sufficient for most applications thanks to the simplicity of LR. Intuitively, because LR is like plotting a line, we only need two points to be able to draw the line if it is in 2D. Because of noise and features not being perfectly correlated with latency, we need more samples.



(a) NN-G



(b) NN-T

Fig. 8: NN models for Xapian.

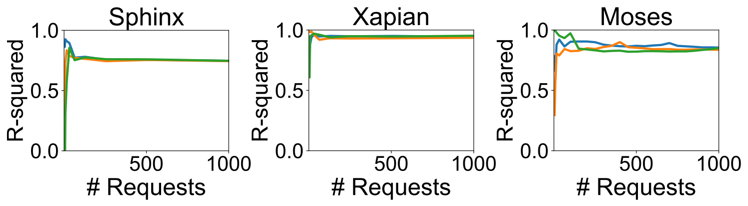


Fig. 9: Sensitivity to training dataset size. Each line represents one trial, and we plot three trials for each application.

RPS is usually less than 10. When an application has started running, we use live data to maintain the latest 1000 samples under each frequency setting in case retraining is needed. At runtime, for every request, ReTail provides the latency prediction at arrival. Also, it obtains its actual service time after completion, replacing the oldest request sample in the dataset for future retraining.

D. Model Retraining

To handle a dynamic environment with interference and system noise, it is important to manage model drift [45] by monitoring and retraining the model online. Cloud environments have three main sources of model drift:

- **Application changes:** LC services have high workload churn, with code roll-outs on a daily/weekly basis [12]. Upon each code update, ReTail checks whether the previously selected features for a given application still correlate with request service time. If not, ReTail's feature selection is triggered again to reselect features. Note that changes in an application's load do not affect request service time, since a request will usually not be interrupted during execution, as introduced in Section III-A (we leave request preemption to future work). Therefore, the latency predictor is robust to fluctuating input load.
- **Application interference:** Cloud providers typically schedule multiple applications on the same physical node to improve server utilization and resource efficiency [13, 35, 38]. Each application is typically deployed in a container/VM with varying amounts of cores, memory, etc. When resource allocation of an application changes[13, 19, 35, 38], or when colocated applications share hardware resources causing resource interference, service time can change, sometimes significantly [13].
- **System interference:** Besides applications, some system tasks or daemons may also run (periodically) on the server, causing additional latency variability to LC applications.

To detect model drift, we monitor the predictor's accuracy at runtime. If RMSE/QoS increases by more than 5%, retraining will be triggered using the latest dataset. The threshold is set to avoid constant retraining that may not help improve model accuracy, e.g., due to small, transient system noise. This threshold is decided based on the measured variation of RMSE/QoS when the application and the system are both running at a stable state, and can be adjusted as needed.

The ReTail manager is in parallel with application threads (Section VI-B), so the retraining overhead does not affect request latency; the old model is used until the new one is available. Nevertheless, retraining overhead is very small.

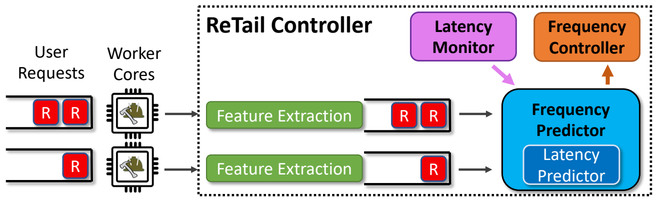


Fig. 10: Overview of ReTail's runtime power management.

E. Live Inference

Inference happens continuously at runtime. The predictor takes in an application name, a target frequency, all the feature values, and then outputs the predicted service time under the target frequency. Inference overhead is extremely small, only $5\mu s$ under LR (also shown in Table IV). Feature values are logged in shared memory (/dev/shm), which incurs marginal overheads. Application and the predictor are communicated through inter-process communication in shared memory. LR models are very small, e.g., if the model is $f(x) = ax + b$, then we store a and b in an array (in L1 cache most likely).

VI. Runtime Power Management

We now describe ReTail's power management system that leverages request-level latency prediction to minimize the power consumption of LC applications while meeting QoS.

A. Feature Extraction

LC applications are usually multithreaded. As shown in Fig. 10, each thread of an LC application typically has a separate application queue to hold requests (the leftmost queues in Fig. 10). Requests in each queue are served first-come-first-serve. Once a request is scheduled, it runs to completion.

Feature extraction extracts all the feature values needed to predict the request service time. This step is unnecessary if predicting latency of only the current request. However, we show in Section VI-B that ReTail needs to predict latency of all the enqueued requests, which may not expose their feature values until after execution. Therefore, ReTail splits each application into two stages, one before and one after the feature values have been obtained, and adds a shallow queue between them. For instance, for Xapian, `processRequest()` is split into two functions, one before and one after `mset` is obtained, and a queue is added between these two functions. The first

Algorithm 1: Frequency Predictor.

```
$CurTime = gettime()$;   // Get the current wall time
for $f = 1..M-1$ do
    // Enumerate from the lowest to second highest frequency
    $Service\_Sum = 0$;   // Accumulated service time
    $ok = true$;   // A flag that indicates if $Freq_f$ is enough.
    for $i = 1..N$ do
        // Predict service time for each requests in the queue under
            the enumerated frequency $Freq_f$,
        $ServiceTime$ = LatencyPredictor($Freq_f$, $Feature_i$);
        // Calculate queueing delay based on the arrival time and
            cumulated service time of all previous requests
        $Queuing = CurTime - Arrival_i + Service\_Sum$;
        $SojournTime = Queuing + ServiceTime$;
        if ($SojournTime > QoS'$) then
            // The request latency budget is not met.
            $ok = false$;
            break;
        // Accumulate the service time of $R_i$ for queuing delay
            estimation of later requests
        $Service\_Sum+ = ServiceTime$;
    if ok then
        // All queued requests can meet their latency budget
        return $Freq_f$;
// None of the lower frequencies is enough
return $Freq_M$;
```

stage, *feature extraction*, simply extracts the arrival time and feature values of each request, and then enqueues requests waiting for further processing. This allows requests to have their features parsed as soon as they arrive, without waiting for all previous requests to finish. Contrary to the belief that adding a queue would unnecessarily prolong sojourn latency, the added queue does not increase the overall queueing delay, but rather moves part of the queueing delay from the original queue (left queues in Fig. 10) to the newly-added one. Since features are obtained before/early during request processing, *feature extraction* is very lightweight; requests quickly pass through the first stage, and wait in the second queue instead.

The same worker cores execute both the first and second application stage, always prioritizing the first stage. Upon any arriving request, even when the core is busy in stage two, it will get interrupted and process stage one of the arriving request, to obtain the features of pending requests. Note that we only observe such interruptions at high load, which also partly explains our diminishing gain at high load in evaluation (Fig. 11a).

B. Frequency Predictor

Once a request is scheduled to run, the frequency predictor (Algorithm 1) is triggered to decide the appropriate frequency for that request. .In the outer loop, we enumerate all the available frequencies, and stop when the minimum satisfying frequency for the current (oldest) request R_1 is found. In the inner loop, we visit all the requests currently in the queue, and check whether each request can meet the latency target under the enumerated frequency. Queueing delay is calculated by accumulating the service time of previously-queued requests. We describe the relationship between a request's latency target (QoS') and the application's QoS target (QoS) in Section VI-C.

Despite the final output being R_1's operating frequency, it is insufficient to examine only R_1. All currently queued requests from R_1 to R_N should be examined. This is because service time of R_1 propagates as queuing delay to $R_2 ... R_N$. Barely meeting QoS of R_1 may leave little room to meet QoS of later requests. Furthermore, ReTail keeps monitoring the request queue. Upon any new requests added before R_1 completes, Algorithm 1 is invoked to check or update R_1's frequency.

To adjust frequency for a specific request, we adjust the frequency of the corresponding core running the request; each thread is pinned to a different core using `taskset` [8]. We use the ACPI frequency driver with the "userspace" governor to allow manual settings of per-core frequencies.

The ReTail runtime is deployed on a dedicated core, to avoid contending with application threads, and to avoid making the frequency prediction and frequency adjustment be on the critical path (Section VII-F evaluates the overhead). Despite the small overhead, the frequency prediction is concurrent with requests; requests by default run at the maximum frequency before the frequency predictor computes the target frequency and the new frequency takes effect.

C. Latency Monitor

The frequency predictor is designed to bring all requests' latency under QoS. However, there are two scenarios when this is not desired or sufficient. First, since QoS is defined for a given percentile such as the 99^{th} percentile, by definition, it allows 1% of requests to violate QoS. If ReTail were to bring 100% of requests under the QoS target, more power would be consumed than necessary. Second, during high load or load spikes, even the highest frequency can be insufficient to meet QoS. To account for unanticipated QoS violations, ReTail needs to be more proactive in boosting frequency.

As a safeguard for both cases, *latency monitor* in ReTail constantly monitors tail latency and adjusts the internal request latency target (QoS'), used in Algorithm 1. QoS' is initially set to the application's QoS target (QoS). Similar to prior work [13, 29, 35], by monitoring tail latency every 100ms, we compare the measured (m) and target (t) tail latency, and adjust QoS' as follows. If $m < 0.9t$ (this threshold can be adjusted based on the variation degree in tail latency) or $m > t$, we increase or decrease QoS' by 5% at a time. In the worst case of sudden load spikes, QoS' can be reduced from 100% to 0% of QoS in $2s$ thanks to the fine-grained monitoring every 100ms, running all the requests at the maximum frequency until the load recovers. The thresholds can be adjusted based on the frequency and degree of load fluctuations.

D. Putting It All Together

When a request arrives at a worker core, *feature extraction* extracts all the feature values needed to predict the request service time. Once scheduled, the frequency predictor decides the operating frequency for the core running the current request, leveraging the latency predictor that outputs predicted request latency under various frequencies. The final frequency is enforced using the system's frequency controller. The latency monitor constantly monitors the application's tail latency, and adjusts the aggressiveness of the frequency predictor, to meet the application's QoS target.

VII. Evaluation

A. Methodology

We evaluate ReTail on an Intel Xeon Gold 6152 CPU with 2 sockets, 22 cores each, and 188GB DDR4, running Ubuntu 16.04 (kernel 4.14). We use the ACPI frequency driver with the "userspace" governor [13] to allow user-defined frequency settings ranging from 1GHz to 2.1GHz in 0.1GHz increments. As frequencies can only be controlled on a per-physical-core (not per-logical-core) basis, we turn off Hyper-Threading [37] to show the maximum potential of ReTail. When Hyper-Threading is enabled, the core frequency can be set to the maximum of the target frequency of all the hyperthreads.

We focus on single-node experiments, as ReTail is an intra-node controller. ReTail can be installed on every node in a datacenter to manage the services running on each node, so the conclusions would still hold for larger-scale evaluation platforms. When interactions between nodes exist (e.g., for multi-tier applications, or services with fanout), where the application only has an end-to-end QoS target, the cluster scheduler which has global system visibility is responsible for determining the per-node QoS target for each service, which ReTail uses to manage power.

Applications are introduced in Table II. We use the open-loop load generator provided by Tailbench as clients. Request inter-arrival times follow an exponential distribution [32] to simulate a Poisson process in which requests are sent independently from one another, at an average rate that remains constant. Client and server threads run on socket0 and socket1, respectively. In socket1, we reserve one core for the OS and one core for the power manager. Both cores always run at the maximum frequency. Applications run on the remaining 20 cores.

We use CPU Energy Meter [4] to measure the energy consumption of the entire socket1, which includes energy consumption from both the package and DRAM, and from both the applications and the power management runtime.

We first evaluate single applications at constant loads, and then evaluate application colocation that incurs online retraining. Each application is instantiated with 20 threads. We define *max load* of each application as the maximum request-per-second (RPS) under QoS when running on the default system. 100% of max load usually takes 60%-80% CPU utilization. Then, we sweep the input load in 10% increments from 10% to 100% of the max load. We run three test trials, each with 60s of warmup and 300s of execution, and record socket1's average power consumption across all trials. We compare with two state-of-the-art latency-based power managers:

- **Rubik** [29] uses statistical modeling and the current queue length to estimate the request latency distribution, and sets each request's frequency based on the estimated tail.

TABLE V: Root Mean Square Error (RMSE) – msec.

	Masstree	ImgDNN	Sphinx	Xapian	Moses	Shore	Silo
Rubik	0.05	0.9	2500	2.8	47.1	3.9	0.5
Gemini	0.03	0.8	217	3.6	3.6	2.2	0.2
ReTail	0.04	0.8	217	0.3	3.6	0.3	0.1

- **Gemini** [51] uses NN for request-level latency prediction. However, Gemini is designed for search engines, and does not include a generalizable process for selecting features and adjusting the model to other services. Therefore, we implement a generalized version of Gemini, which uses all available features at request arrival as input features, and follows the steps in Section V-A to tune the NN structure.

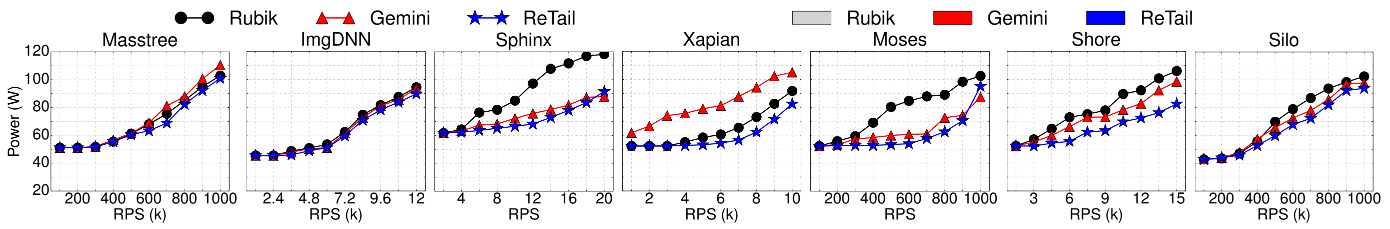
B. Power Consumption

Fig. 11a shows the average power consumption, and Table V shows the RMSE for each power manager.

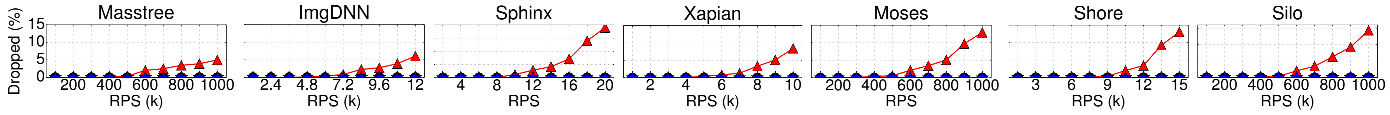
Rubik uses the estimated latency distribution's tail to calculate the target frequency. Because the actual latency is usually smaller than the tail, RMSE under Rubik is the largest, leading to conservative frequency adjustment and power saving.

Gemini and ReTail reduce power over Rubik by leveraging request features to perform request-level latency prediction, improving prediction accuracy. Compared with Rubik, Gemini has three major drawbacks:

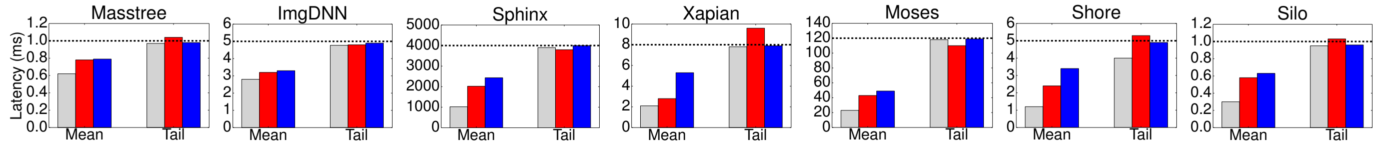
1) Gemini's power management algorithm drops all requests that are predicted to miss the deadline. The percentage of dropped requests increases super-linearly with load (Fig. 11b), reaching up to 16% (average 9.2%) at max load. Due to the unique characteristic of the search workload that Gemini targets, multiple requests collectively contribute to a search query; the search aggregator can still form a response even with some dropped requests (though with degraded search quality). However, this characteristic is not generalizable to other LC applications. Drop rate directly affects user experience [12], and should be kept as low as possible. ReTail does not drop requests. Since no work needs to be done for dropped requests and energy can be saved naturally, ReTail sometimes consumes more power than Gemini at high load. For instance, at RPS=20 for Sphinx, Gemini drops 16% of requests, i.e., Gemini does roughly 16% less work than ReTail, but consumes only 3.5% less power. In most cases, Gemini consumes more power while also dropping many requests. This is mainly due to two inefficiencies in the power management algorithm. First, Gemini assumes that requests are 100% compute-intensive; the target frequency is calculated based on the estimated number of cycles and the time budget. Ignoring memory cycles that cannot be changed by adjusting CPU frequency, Gemini tends to overestimate frequency, which saves less power. Second, Gemini's two-step DVFS adjusts almost every request's frequency twice, one low initial frequency and, later, one high frequency to meet the deadline in case of prediction errors. As power increases super-linearly with frequency, this consumes more power than ReTail's "one-step" approach that sets a single frequency for



(a) Power consumption under each power manager at various input loads.



(b) Percentage of dropped requests under each power manager at various input load.



(c) Mean and tail latency under each power manager at max load. The horizontal dotted lines are the QoS targets.

Fig. 11: Comparison between Rubik, Gemini, and ReTail.

each request most of the time. ReTail, instead, relies on the latency monitor to adjust the internal request latency target, to combat prediction errors.

2) Gemini's feature space include only features that are readily available at request arrival. For applications that require applications features, such as Xapian, Gemini consumes even more power than Rubik. ReTail opens up the feature space to include features that are not necessarily available at request arrival, resulting in features that correlate better with request service time for Xapian and Shore, which eventually leads to much lower prediction errors (RMSE) than Gemini. The difference between Gemini and ReTail is smaller for Shore because only two request types in Shore require features, so Gemini is still beneficial for the other two request types. Like Shore, two of Silo's request types also require application features, but the difference between the three power managers for Silo is even smaller. This is because Silo's request latency is in the sub-millisecond granularity, and the overhead to adjust frequency (100s of microseconds) is non-negligible compared to request latency (Section VII-F), which makes per-request frequency adjustment less effective.

3) Gemini's high inference overheads (more than $300\mu s$ per request as shown in Table IV) are significant in applications with sub-millisecond request latencies (i.e., Masstree and Silo). For Masstree, Gemini consumes more power than Rubik due to the additional inference overhead.

In short, ReTail reduces power consumption by up to 36.2% and 35.6% (average 11.5% and 8.9%) compared with Rubik and Gemini, respectively, without dropping any requests.

C. *QoS Awareness*

QoS is always ReTail's first priority; power is saved only under QoS satisfaction. The latency monitor (Section VI-C) is especially designed to adjust the internal request latency target to meet the overall application's QoS target even under high load, or with prediction errors, and/or upon model drift.

As shown in Fig. 11c, even under high load, ReTail still meets QoS. Compared with Rubik, ReTail has higher mean latency due to more accurate identification of short requests, which are slowed down. Rubik also always meets QoS due to the conservative latency prediction and frequency adjustments. However, Gemini violates QoS for Xapian, Shore and Silo (i.e., those that need application features) due to prediction errors, and for Masstree and Silo due to long inference time. In addition, Gemini simply sets the request latency target to the application's QoS target, i.e., $QoS' = QoS$. This is especially problematic for high load, when frequency has to be proactively boosted to avoid queues getting quickly filled up in the future. Gemini is also oblivious to how QoS is defined, i.e., the power manager will perform exactly the same when tail latency is defined based on the 90^{th} or 99^{th} percentile.

D. *ReTail Decomposition*

To understand the effectiveness of each of the three components in ReTail, i.e., ReTail's feature selection, latency prediction and power management, we decompose ReTail and compare with mechanisms that differ in one or more of the components. Specifically, for feature selection, as shown in Fig. 12, dotted lines use only request features for feature selection (adopted by Adrenaline and Gemini), and solid lines use both request and application features (adopted by ReTail). For each feature space, we adopt five different combinations of prediction and power management algorithms, including

- **Adrenaline**: classification-based prediction and power management;

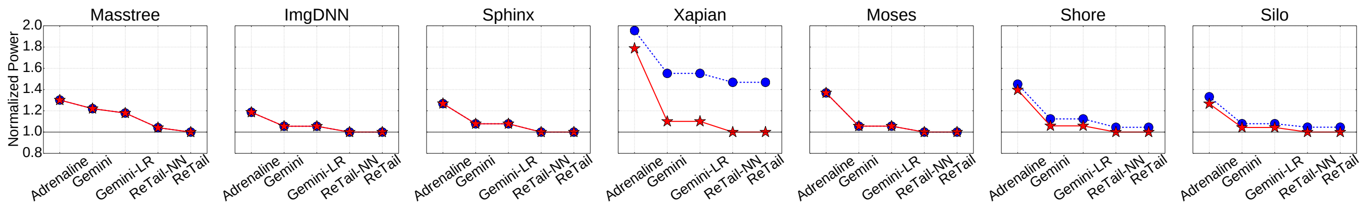


Fig. 12: Normalized power consumption under two different feature space (dotted/solid lines use request/request+application features), and five combinations of prediction and power management algorithms, when each application is at 70% of max load.

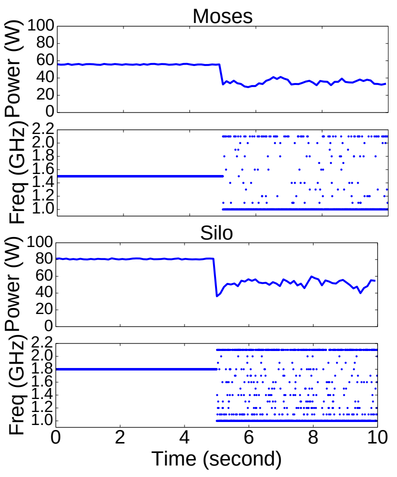


Fig. 13: Colocating Moses and Silo. PARTIES alone manages resources in the first 5s. After 5s, ReTail is applied on top of PARTIES to further reduce power consumption.

- **Gemini/Gemini-LR**: NN/LR-based prediction and Gemini's two-step DVFS;
- **ReTail-NN/ReTail**: NN/LR-based prediction and ReTail's power management algorithm.

Fig. 12 shows power consumption that is measured when each application runs at medium high load (i.e., 70% of max load), normalized to the power consumption under ReTail with both request and application features as feature space.

First, comparing dotted and solid lines shows that increasing the feature space to include application features reduces power consumption for Xapian, Shore, and Silo (the last two categories in Section III-D). Second, Adrenaline leads to up to 80% more power consumption comparing to ReTail, which justifies the need for latency-based prediction. Third, comparing Gemini and Gemini-LR or ReTail-NN and ReTail, the difference is negligible. This shows that LR is enough for latency prediction. Finally, ReTail's power management outperforms Gemini's two-step DVFS.

E. ReTail under Colocation

ReTail manages a single resource, power, for a single LC application. When multiple applications are colocated on the same node, one ReTail runtime is installed for each colocated application, managing its core frequency, while resource managers such as PARTIES [13] can be deployed to manage other shared resources. Fig. 13 shows the synergy between

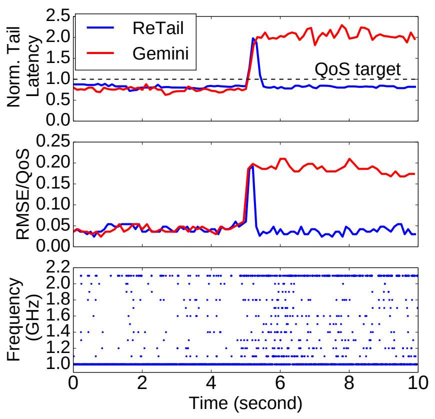


Fig. 14: Tail latency, prediction accuracy, and core frequency with time under ReTail and Gemini. Moses first runs alone, and then is colocated with a batch job starting at 5s.

ReTail and PARTIES. Two LC applications, Moses and Silo, are colocated on the same node, each running at 50% of their respective *max load*. PARTIES first finds a feasible resource allocation such that both Moses and Silo meet QoS. However, because PARTIES manages power at application-granularity, i.e., all application threads are set the same frequency (first 5s in Fig. 13), power consumption is suboptimal. ReTail is applied over PARTIES during PARTIES' *downsize* phase, to further reduce power consumption. At 5s, the ReTail runtime is triggered, managing frequency at the request granularity, leading to more than 30% of power savings.

F. ReTail Overhead

All experiments take all overheads into account, including:

- The time for the frequency predictor to calculate the target frequency. Since the number of inferences ($5\mu s$/inference) per request varies with queue length, it takes 5-100μs (average 25μs) to calculate the final frequency per request.
- The time for a new frequency to take effect. Upon a frequency adjustment, the new frequency value is written to the frequency MSR [5]. It takes an average of $5\mu s$ to write the register. However, measurement of real-time CPU frequency via `i7z` [7] shows that it takes 10-500μs (average 200μs) for the new frequency to take effect.

G. Online Retraining for Model Drift

ReTail continuously monitors the prediction accuracy and triggers model retraining when model drift is detected (Section V-D). To evaluate ReTail's responsiveness to model drift, we first run an LC application, Moses at 20% of its max load for 5s, and then introduce application interference by colocating Moses with a batch job on the same socket. The batch job is constructed by mixing compute-intensive and memory-intensive microbenchmarks [13, 18], 5 threads each.

Fig. 14 shows the real-time tail latency of Moses, the latency predictor's accuracy (RMSE/QoS value), and the frequency of a core running Moses over time. Initially, Moses runs alone without resource contention, and cores are sparsely boosted to higher frequencies. At 5s, the batch job arrives, splitting all cores and LLC cache ways with Moses, each taking 10 cores and 10 cache ways. Due to the reduced resource allocation, Moses experiences QoS violations, but tail latency under ReTail quickly recovers after less than 0.5s thanks to the immediate detection of model drift: at 5s, RMSE/QoS increases by 15%, triggering model retraining which takes less than 0.1s. The latency predictor is quickly updated afterwards, shown in the recovered RMSE/QoS value. Compared to the first 5s when Moses runs alone, the cores running Moses in the latter 5s have to spent more time at higher frequencies to combat the reduced resources. In contrast, Gemini has longstanding QoS violations. It does not detect and adapt to environment changes, and even if it does, the large training overhead prohibits it from reacting as quickly as ReTail.

VIII. Conclusion

We have presented ReTail, a QoS-aware power manager for latency-critical applications using request-level latency prediction. ReTail systematically selects the features that best correlate with request latency for a general LC application, and builds a latency predictor using linear regression. ReTail achieves an average of 8.9% (up to 35.6%) power savings compared with the best related work, while meeting QoS.

Acknowledgements

This work was supported in part by NSF and the Semiconductor Research Corporation (SRC) through the DEEP3M Center, part of the E2CDA program; and by DARPA and SRC through the CRISP Center, part of the JUMP program. Christina Delimitrou was partially supported by a Sloan Foundation Research Award, a Microsoft Research Faculty Fellowship, an Intel Rising Star Award, a Google Faculty Research Award, and NSF grants NeTS CSR-1704742 and CCF-1846046. The authors wish to thank the COE/CIS/Tech Information Technology Support Group, in particular Michael Woodson, for their technical assistance.

References

[1] "Apache lucene," https://lucene.apache.org.
[2] "Coefficient of determination," https://en.wikipedia.org/wiki/Coefficient_of_determination.
[3] "Correlation ratio wikipedia," https://en.wikipedia.org/wiki/Correlation_ratio.
[4] "Cpu energy meter," https://github.com/sosy-lab/cpu-energy-meter.
[5] "Cpu frequency scaling," https://wiki.archlinux.org/index.php/CPU_frequency_scaling.
[6] "El pas," https://elpais.com/.
[7] "i7z: Frequency reporting tool for linux," https://github.com/ajaiantilal/i7z.
[8] "Linux taskset," https://linux.die.net/man/1/taskset.
[9] "Multiple correlation wikipedia," https://en.wikipedia.org/wiki/Multiple_correlation.
[10] "Pearson correlation coefficient wikipedia," https://en.wikipedia.org/wiki/Pearson_correlation_coefficient.
[11] A. Acero, Acoustical and environmental robustness in automatic speech recognition. Springer Science & Business Media, 2012, vol. 201.
[12] L. A. Barroso, U. Hölzle, and P. Ranganathan, "The datacenter as a computer: Designing warehouse-scale machines," Synthesis Lectures on Computer Architecture, vol. 13, no. 3, pp. i–189, 2018.
[13] S. Chen, C. Delimitrou, and J. F. Martínez, "PARTIES: QoS-aware resource partitioning for multiple interactive services," in International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2019.
[14] S. Chen, S. GalOn, C. Delimitrou, S. Manne, and J. F. Martínez, "Workload characterization of interactive cloud services on big and small server platforms," in International Symposium on Workload Characterization (IISWC), 2017.
[15] R. Cochran, C. Hankendi, A. K. Coskun, and S. Reda, "Pack & cap: adaptive dvfs and thread packing under power caps," in 44th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2011.
[16] T. P. P. Council, "TPC-C benchmark, revision 5.11," 2010.
[17] J. Dean and L. A. Barroso, "The tail at scale," Communications of the ACM, vol. 56, no. 2, pp. 74–80, 2013.
[18] C. Delimitrou and C. Kozyrakis, "iBench: Quantifying interference for datacenter workloads," in 2013 IEEE International Symposium on Workload Characterization (IISWC). Portland, OR, September 2013.
[19] C. Delimitrou and C. Kozyrakis, "Quasar: Resource-efficient and qos-aware cluster management," in Proceedings of the Nineteenth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), 2014.
[20] C. Delimitrou and C. Kozyrakis, "Bolt: I Know What You Did Last Summer... In The Cloud," in Proceedings of the Twenty Second International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), April 2017.
[21] L. Deng, "The mnist database of handwritten digit images for machine learning research [best of the web]," IEEE Signal Processing Magazine, vol. 29, no. 6, pp. 141–142, 2012.
[22] A. S. Goldberger, "Classical linear regression," Econometric theory, pp. 156–212, 1964.
[23] M. E. Haque, Y. He, S. Elnikety, T. D. Nguyen, R. Bianchini, and K. S. McKinley, "Exploiting heterogeneity for tail latency and energy efficiency," in 50th International Symposium on Microarchitecture (MICRO), 2017.
[24] C.-H. Hsu, Y. Zhang, M. A. Laurenzano, D. Meisner, T. Wenisch, J. Mars, L. Tang, and R. G. Dreslinski, "Adrenaline: Pinpointing and reining in tail queries with quick voltage boosting," in IEEE 21st International Symposium on High Performance Computer Architecture (HPCA), 2015.
[25] E. İpek, O. Mutlu, J. F. Martínez, and R. Caruana, "Self-optimizing memory controllers: A reinforcement learning approach," in International Symposium on Computer Architecture (ISCA), 2008.
[26] C. Isci, A. Buyuktosunoglu, C.-Y. Cher, P. Bose, and M. Martonosi, "An analysis of efficient multi-core global power management policies: Maximizing performance for a given power budget," in 39th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2006.
[27] G. James, D. Witten, T. Hastie, and R. Tibshirani, An introduction to statistical learning. Springer, 2013, vol. 112.
[28] M. Jeon, S. Kim, S.-w. Hwang, Y. He, S. Elnikety, A. L. Cox, and S. Rixner, "Predictive parallelization: Taming tail latencies in web search," in 37th international ACM SIGIR conference on Research & development in information retrieval, 2014.
[29] H. Kasture, D. B. Bartolini, N. Beckmann, and D. Sanchez, "Rubik: Fast analytical power management for latency-critical systems," in 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 2015.
[30] H. Kasture and D. Sanchez, "Tailbench: a benchmark suite and evaluation methodology for latency-critical applications," in IEEE International Symposium on Workload Characterization (IISWC), 2016.
[31] S. Kim, Y. He, S.-w. Hwang, S. Elnikety, and S. Choi, "Delayed-dynamic-selective (dds) prediction for reducing extreme tail latency in web search," in Proceedings of the Eighth ACM International

Conference on Web Search and Data Mining, 2015.
[32] J. Leverich and C. Kozyrakis, "Reconciling high server utilization and sub-millisecond quality-of-service," in Proceedings of the 9th European Conference on Computer Systems, 2014.
[33] J. Li, N. K. Sharma, D. R. Ports, and S. D. Gribble, "Tales of the tail: Hardware, os, and application-level sources of tail latency," in the 2014 ACM Symposium on Cloud Computing (SoCC), 2014.
[34] D. Lo, L. Cheng, R. Govindaraju, L. A. Barroso, and C. Kozyrakis, "Towards energy proportionality for large-scale latency-critical workloads," in 41st International Symposium on Computer Architecture (ISCA), 2014.
[35] D. Lo, L. Cheng, R. Govindaraju, P. Ranganathan, and C. Kozyrakis, "Heracles: Improving resource efficiency at scale," in Proceedings of the 42nd Annual International Symposium on Computer Architecture (ISCA), 2015.
[36] C. Macdonald, N. Tonellotto, and I. Ounis, "Learning to predict response times for online query scheduling," in Proceedings of the 35th international ACM SIGIR conference on Research and development in information retrieval (SIGIR), 2012.
[37] D. T. Marr, F. Binns, D. L. Hill, G. Hinton, D. A. Koufaty, J. A. Miller, and M. Upton, "Hyper-threading technology architecture and microarchitecture." Intel Technology Journal, vol. 6, no. 1, 2002.
[38] J. Mars, L. Tang, R. Hundt, K. Skadron, and M. L. Soffa, "Bubble-up: increasing utilization in modern warehouse scale computers via sensible co-locations," in Proceedings of the 44th IEEE/ACM International Symposium on Microarchitecture (MICRO), 2011.
[39] D. Meisner, C. M. Sadler, L. A. Barroso, W.-D. Weber, and T. F. Wenisch, "Power management of online data-intensive services," in 38th Annual International Symposium on Computer architecture (ISCA), 2011.
[40] A. Paszke, S. Gross, F. Massa, A. Lerer, J. Bradbury, G. Chanan, T. Killeen, Z. Lin, N. Gimelshein, L. Antiga et al., "Pytorch: An imperative style, high-performance deep learning library," arXiv preprint arXiv:1912.01703, 2019.
[41] R. A. Popa, C. M. S. Redfield, N. Zeldovich, and H. Balakrishnan, "Cryptdb: Protecting confidentiality with encrypted query processing," in Proceedings of the Twenty-Third ACM Symposium on Operating Systems Principles, ser. SOSP '11. New York, NY, USA: Association for Computing Machinery, 2011, p. 85100.
[42] N. Samadric, "Tbd," in Proceedings of the 54th IEEE/ACM International Symposium on Microarchitecture, 2021.
[43] H. Sasaki, S. Imamura, and K. Inoue, "Coordinated power-performance optimization in manycores," in 22nd International Conference on Parallel Architectures and Compilation Techniques (PACT), 2013.
[44] W. Shang, M. Nagappan, and A. E. Hassan, "Studying the relationship between logging characteristics and the code quality of platform software," Empirical Software Engineering, vol. 20, no. 1, pp. 1–27, 2015.
[45] P. Siva and T. Xiang, "Weakly supervised object detector learning with model drift detection," in 2011 International Conference on Computer Vision (ICCV), 2011.
[46] A. Sriraman and A. Dhanotia, "Accelerometer: Understanding acceleration opportunities for data center overheads at hyperscale," in Proceedings of the Twenty-Fifth International Conference on Architectural Support for Programming Languages and Operating Systems, ser. ASPLOS '20. New York, NY, USA: Association for Computing Machinery, 2020, p. 733750.
[47] W. Wang, Y. Hu, L. Chen, X. Huang, and B. Sunar, "Exploring the feasibility of fully homomorphic encryption," IEEE Transactions on Computers, vol. 64, no. 3, pp. 698–706, 2015.
[48] S. Wright, "Correlation and causation," J. agric. Res., vol. 20, pp. 557–580, 1921.
[49] H. Yang, A. Breslow, J. Mars, and L. Tang, "Bubble-flux: precise online QoS management for increased utilization in warehouse scale computers," in Proceedings of the 40th International Symposium on Computer Architecture (ISCA), 2013.
[50] D. Yuan, S. Park, and Y. Zhou, "Characterizing logging practices in open-source software," in the 34th International Conference on Software Engineering (ICSE), 2012.
[51] L. Zhou, L. N. Bhuyan, and K. K. Ramakrishnan, "Gemini: Learning to manage cpu power for latency-critical search engines," in Proceedings of the 53rd IEEE/ACM International Symposium on Microarchitecture (MICRO), 2020.