# At What Cost?
# Enhanced Cost Functions for Fair LLM Serving

Karthik Vetrivel, Cary Xiao, Michael Rybalkin, and Jamie Ullman

Department of Computer Science, Stanford University

March 2026

## Abstract

LLM providers receive requests of vastly different sizes and frequencies from a diverse array of clients. To ensure that no client is starved in responding to those requests, Sheng et al. explore a metric for fairness in "Fairness in Serving Large Language Models" (OSDI '24) [6]. Using Virtual Tokens as a proxy for time, the paper presents the Virtual Token Counter (VTC) scheduler, an algorithm for the LLM scheduler that ensures fairness without artificially causing under-utilization. In fact, they prove a "2x tight upper bound on the service difference between two backlogged clients" [6]. "At What Cost?" presents a comprehensive replication and extension of said VTC scheduler to incorporate the needs of the present day. We first independently verify the fairness guarantees of the VTC algorithm, confirming its efficacy in fairly responding to multiple clients from a system at capacity. We then reexamine VTC's existing use of a linear cost function, which currently assumes a constant time cost per token for any request size. Due to the nature of attention, the computational cost of processing a request is expected to increase quadratically as the context window grows. In this paper, we explore the use of a quadratic cost function to provide a more accurate measure of fairness by today's standards. We additionally stress the system with synthetic workloads matching realistic divergences in context windows. Our results demonstrate that while VTC Fairness as previously understood remains robust, redefining the cost definition maintains a reduced service difference while being both a more accurate measure of the subjective experienced "fairness" for a particular client as well as more resistant to asymmetric workloads.

## 1 Introduction

The use of LLMs has proliferated every facet of our modern society. Naturally, this means that LLM providers are servicing a massive heterogeneous user base with client requests ranging drastically both in terms of size and frequency [4]. In recent years, the compute volume has increased to meet the request volume by doubling approximately every 6 months, representing a level of exponential growth even mind-boggling by Moore's standards [5]. And further, meeting this rising demand is imperative as the world shifts to rely on AI as though it were a utility [4]. Just as utility abstractions such as the internet and electricity must consider fairness, and in particular, avoid starving any particular client, LLM services too must take on this concern. Yet, it was not until the Virtual Token Counter (VTC) scheduler was introduced by Sheng et al. (OSDI '24) that we had a solution to ensure fairness in serving these various requests [6].

Exploring the unique aspects of LLMs, Sheng et al. show that a new definition and understanding of fairness must be used in the context of LLMs than those that we currently use in traditional contexts such as networking and operating systems. They attribute this to three primary differences. First, "packet" lengths are unknown before they are scheduled due to the varying output token lengths. Second, the token processing cost is not fixed, as the marginal cost per token increases with token count due to the quadratic nature of attention. Third, the processing rate of the LLM is not fixed, for the same reason. As a result, these three fundamental assumptions in traditional contexts of fairness break down. A traditional workaround given these constraints is simply to rate limit each client, though this can cause an underutilization of the LLMs if a client is being capped despite there being available capacity in the network. This leads to wasted batched compute resources and slower response times. Sheng et al. introduce a fair scheduling algorithm that can be baked into a scheduler in an LLM service, such as S-LoRA, that addresses these concerns by considering the cost of a client's prior requests in deciding which client's request to serve next. They demonstrated that we can ensure fair servicing of requests over time without needing to rate limit.

To address this concern, we replicate the contributions of the VTC algorithm to independently validate the fairness that it delivers. Building on this baseline, we show that using a quadratic cost function delivers equivalently low absolute service difference between clients, but with a definition of "service" more closely tuned to the actual compute cost (Figure 5 in the appendix). It is critical to note this nuance. In this paper, we introduce a new definition of "service", as service is measured by the cost function selected. So, by achieving a low absolute service

difference with our new cost function, we show that the VTC algorithm continues to deliver theoretical fairness with our new model all while delivering what we consider to be a more accurate definition of fairness given modern day request patterns. Moreover, by leveraging synthetic workloads and more modern, higher-performance NVIDIA H100 GPUs with much higher throughput than those used in the original VTC paper, we demonstrate that the quadratic cost function proves to be a more resilient model of cost in the modern day.

# 2 Background

In this section we provide a fundamental understanding of how LLMs process requests to motivate our introduced cost function and illuminate where our updated VTC algorithm fits in.

When a client makes a **request**, said request is sent to an HTTP server that handles the incoming and outgoing connections to the LLM. The request is then tokenized, a process by which natural language is split and vectorized into representative mathematical chunks. These **tokens** are fed to the LLM as **input tokens**, and the LLM enters the **prefill** stage. During this first of two stages, the LLM processes the entire input sequence in parallel to compute the initial **Key-Value (KV) cache**, which stores the attention states for every input token. Once this cache is populated, the LLM enters a **decode** stage where it generates tokens one-by-one (auto-regressively). Each new token is based on the trained model weights, the existing KV cache, and the string of tokens previously processed/generated in that session (including input tokens). These resulting tokens are called the **output tokens** and are then de-tokenized back into natural language, returned to the HTTP server, and sent to the client as the response to their original message.

Because the tokens are the fundamental unit of computation in Transformers, they serve as a robust proxy for resource consumption of a request. To estimate this cost, the original VTC paper [6] uses a linear heuristic: $Cost = w_{\text{in}} \cdot L_{\text{in}} + w_{\text{out}} \cdot L_{\text{out}}$. Using common industry pricing as a baseline, they set $w_{\text{in}} = 1$ and $w_{\text{out}} = 2$, assuming a constant cost per token regardless of total sequence length.

However, the underlying scaled dot-product attention mechanism, the core of the Transformer architecture, is inherently non-linear. In the prefill stage, the model processes all $L_{\text{in}}$ tokens simultaneously to compute the attention matrix, an operation with $O(L_{\text{in}}^2)$ computational complexity. During the decode stage, although the use of a KV cache prevents re-computing previous states, each new token still must attend to every preceding token in the sequence. Consequently, generating the $k$-th output token requires $O(L_{\text{in}} + k)$ operations. When summed over the entire decoding sequence, the total complexity becomes $O(L_{\text{in}} \cdot L_{\text{out}} + L_{\text{out}}^2)$, making the entire cost quadratic relative to the total context length.

Beyond the raw FLOPs, these quadratic effects are exacerbated by memory bottlenecks. As the sequence length grows, the KV cache expands, requiring more time to retrieve attention states from VRAM. While a linear 2x weight for output tokens is a useful heuristic for shorter sequences, it fails to account for the accelerating cost of attention as context windows reach the scales used in modern applications. We argue that a quadratic cost function provides a more accurate measure of the actual service rendered to a client, particularly as the industry shifts towards context windows of $10^5$ to $10^7$ tokens.

Finally, the cost function is leveraged in the **scheduler** for **continuous batching**, the process by which a batch of requests is prepared on every token generation step. Continuous batching enables GPUs to be maximally utilized by injecting a new request into an ongoing batch as soon as a batched request finishes. To enforce real-time fair serving based on a given cost estimate at this request serving point, the VTC algorithm calculates the marginal cost of every additional token generated in the batch and updates a virtual token cost counter attributed to each respective client. As a result, at the start of each execution step, the VTC fair scheduler is able to immediately select the next request from the client that has the lowest counter of clients in the queue. Notably, this is not necessarily the client whose request was received first, demonstrating the mechanism by which requests can be delayed from greedier clients in favor of allowing more starved clients "skip the line", so to speak.

# 3 Replicating VTC

## 3.1 Experiment Rescaling

To independently verify the fairness guarantees of the Virtual Token Counter (VTC) scheduler, we replicated the experiments described by Sheng et al. [6] using the S-LoRA framework. While the original study utilized NVIDIA A10 GPUs (24GB VRAM), our replication was conducted on NVIDIA H100 GPUs (80GB VRAM). The H100 represents a more modern GPU for high-performance LLM serving. Using this hardware allowed us to test the VTC's robustness using significantly higher compute loads and larger batch sizes that would otherwise be memory-constrained on older hardware architectures. Due to the hardware having significantly higher compute capacity, it was necessary to scale up the request rates in the experiments to ensure that the VTC logic is the factor that determines fairness, rather than just the hardware's speed.

The primary challenge in replicating VTC on H100 hardware was properly maintaining a consistent backlog of requests. The H100's high throughput meant the original paper's request rates of 1.5 or 3.0 req/s were processed too quickly, preventing the clients from becoming backlogged. These conditions are inadequate for assessing

fairness under resource contention, as a scheduler with no queue is effectively FCFS. To address this when replicating experiments and plots we scaled up the request rates to 2.0 and 4.0 req/s for experiment 1. By scaling up the request rates of the clients, we successfully introduce a growing backlog as desired, allowing us to observe VTC's prioritization of clients using the virtual counters.

We also scaled the system's capacity limits to match the 80GB VRAM on the H100, increasing `max_total_token_num` to 60,000 (from 6,000) and the `running_max_req_size` from 10 to 16. These changes improved the batching efficiency, keeping the hardware fully utilized during decoding stages.

### 3.2 Infrastructure Optimizations

Our replication efforts revealed two critical system-level bottlenecks that were not present in the low-throughput experiments of the original paper:

- **Connection Pool Starvation**: The default limit of 100 concurrent connections in the `aiohttp` client session caused artificial request throttling. In a backlogged state, requests would queue at the HTTP layer rather than the LLM scheduler. We resolved this by removing the connection limit (`limit=0`), so that the arrival rate accurately reflects the requested workload.
- **In-Flight Request throttling**: To prevent system-wide exhaustion of file descriptors or memory during extreme backlogging scenarios, we introduced a per-client `asyncio.Semaphore`, allowing us to maintain stable pressure on the server queue without crashing the underlying uvicorn service.

### 3.3 Metric Collection and Visualization

A key difference in our replication is the granularity of the data visualization. The original paper uses non-overlapping 30 second windows for calculating service rates. Our plots (e.g. Figure 1) use a 60 second rolling window sampled every 5 seconds (averaging over $[t-30, t+30]$). This provides a smoother representation of the steady state than in the original VTC paper, and better highlights oscillations inherent to token-level scheduling.

We observed that the response time metrics in our replication often appear to truncate before the end of the experiment. This is a side effect of the deep backlog causing requests to time out. Requests sent near the end of the experiment will not have a known response time if they are not addressed before the end of the experiment. Our tests indicate that the Python-side scheduling overhead was negligible compared to the GPU compute time on the H100, confirming that the VTC algorithm's logic is well suited for high-performance serving.

### 3.4 Replicating Figures from VTC Paper

In the paper which proposed VTC [6], each experiment has a corresponding figure showing plots of service rates. We picked several experiments from the original VTC paper to replicate, including the experiments which correspond to Figures 3 and 4 in that paper. In the rest of our paper, we will refer to the Figure 3 experiment from the Sheng et al. VTC paper as "Experiment 1", and the Figure 3 in our work is unrelated to this experiment.

Looking specifically at Figure 1, we see that our independent re-implementation of VTC was able to largely match the behavior seen by the corresponding Figure 3 in [6]. However, because we use the 5-second rolling average and are serving on an H100, the service that each client receives is around 2 times higher, and our calculated service over time is more stable.

## 4 An Improved Cost Function

While the proposed definition of service using a weighted sum of input and output tokens in [6] (which we will refer to as VTC Linear) is sufficient for workloads with relatively similar input and output lengths, VTC Linear presents a significant problem when handling queries of vastly different input or output sizes. As described in Section 2, the computational cost of the prefill and decode stages scales quadratically, meaning that larger queries will require significantly more compute per token than smaller queries.

Because of this imbalance, in this section, we investigate a synthetic workload - akin to those in [6] - that we will refer to as **Experiment 2**. This workload is a variation of Experiment 1, where Client 1 sends 3 queries per second, and each query has an input and output length of 256. Client 2, on the other hand, sends queries with an input and output length of 2048, but drops the query rate to only 0.375 requests per second so that the amount of requested service (as defined by VTC Linear) over time is the same as Client 1. This query size was chosen because it uses the largest tokens per query possible in the Llama-7b model we serve via S-Lora, as the maximum context window size of Llama-7b is 4,096 tokens. Like Experiment 1's workload, both clients are requesting above the capacity of the GPU and thus become bottlenecked over time. While synthetic, this scenario is designed to mimic likely real-life workloads, where a single GPU is required to handle clients running tasks that require large requests (i.e., agentic workloads, essay generation) while other clients run tasks that use small requests (i.e., binary classification).

In this workload, Client 2's longer request lengths mean that the computational cost per token will be significantly higher than Client 1's tokens. However, VTC Linear assigns a constant level of service per token to both Client 1 and Client 2, meaning that Client 2's level of service will be underestimated, which will cause Client

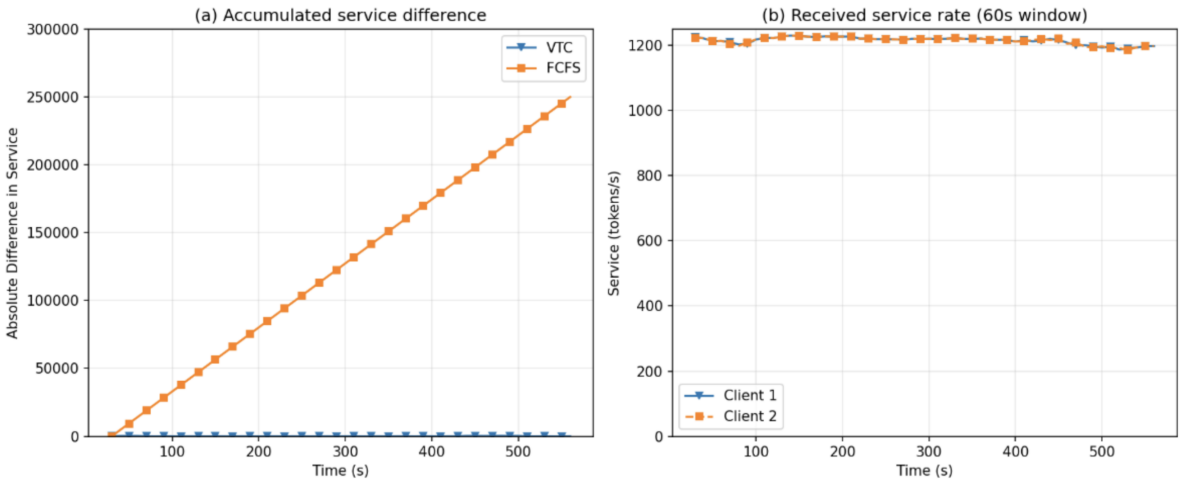


Figure 1: Replicated results from Figure 3 in [6]. Like in [6], (b) specifically shows the received service rate for Client 1 and Client 2 using VTC.

2 to take up a majority of the utilization of the GPU, even though Client 1 and Client 2 have the same level of received service. To demonstrate this behavior, we ran Experiment 2's workload using our original replication of [6], which we will refer to as the VTC Linear scheduler, with the resulting received service rate over time shown in Figure 2. While the service received by Client 1 and Client 2 is nearly identical in this workload, Client 1 experiences approximately 200 less "service" than it did in Experiment 1. This demonstrates that the amount of service received by a given client can thus be affected by the workload that other clients are running.

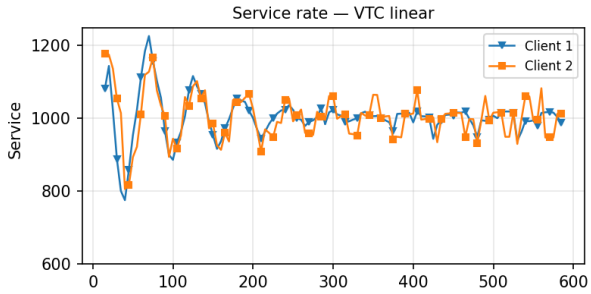


Figure 2: Service received (using VTC Linear definition) when running Experiment 2 using the original VTC scheduler. Comparing this with Figure 1, we see that the steady state of service Client 1 receives is *lower* in this experiment, indicating that Client 2's workload of larger queries here negatively impacts the throughput experienced by Client 1.

While this difference might seem minor when comparing these two workloads, as context windows progressively grow and become near-arbitrarily large, VTC Linear's definition of service could severely underestimate the level of service per token for the largest queries, leading to near-starvation for a client in the case where a client with the largest possible query size is on the same GPU as a client with a small query size. Similar to the argument given by Greenberg et al. in their paper proposing VL2 [3], we believe that all clients should have access to their fair share of hardware resources when backlogged, and that their throughput or allocated share of resources should not be dependent on the workloads other clients are running. As such, the minimum hardware utilization of a given backlogged client should be at least $\frac{1}{n}$, where $n$ is the number of backlogged clients on the machine. To better achieve this goal, we propose a quadratic-based service function, which we will describe in following subsection.

## 4.1 VTC Quadratic

We define a new definition of service, which we will call VTC Quadratic. On top of the weighted sum of tokens in VTC Linear, we add two additional components to the service counter, which scale proportionally with the square of the input and output length, respectively. First, upon choosing the next query to schedule, we add an extra $w_i * \frac{(\text{input length})^2}{d}$ on top of the $w_i * (\text{input length})$ to the given client's service counter. Upon decoding each output token, on top of adding $w_o$, we also add $w_o * \frac{(\text{current output length})}{d}$ to the client's service counter. Via Gauss' formula, the overall service $W$ received by a client $c$ for a given query $q$ with input length $\ell_i$ and output length $\ell_o$ is equal to the following:

$$W_c(q) = w_i\ell_i + w_o\ell_o + \frac{w_i}{d}{\ell_i}^2 + \frac{w_0}{2d}(\ell_i)(\ell_i + 1)$$

Thus, the total service received by a client between times $t_1$ and $t_2$, $W_c(t_1, t_2)$, is now calculated by adding up the service from the prefill step of all queries processed for that client and the service received from all decode tokens generated between $t_1$ and $t_2$. For our experimentation, we

choose $d = 1000$ so that a token with a medium-length query in Llama-7b with 1024 input and output tokens would receive around twice the service per token as a minimum-size query, closely matching the 2x tiered price jump seen in modern LLM providers' API pricing [1] [2].

### 4.2 Evaluation

To evaluate our quadratic service definition and VTC scheduler, we ran Experiment 2 using the Quadratic VTC Scheduler on top of S-LoRA, resulting in Figures 3 and 4 in the Appendix. To further explain these two figures, the main difference between the two lies in how each Y-axis was calculated: using the generated logs from the same 10-minute synthetic workload, we calculate the received service over time using VTC Quadratic for Figure 3 and VTC Linear in Figure 4.

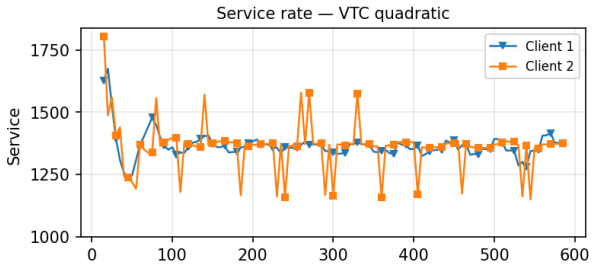


Figure 3: Service received (using VTC Quadratic definition) when running Experiment 2 using the Quadratic VTC scheduler.

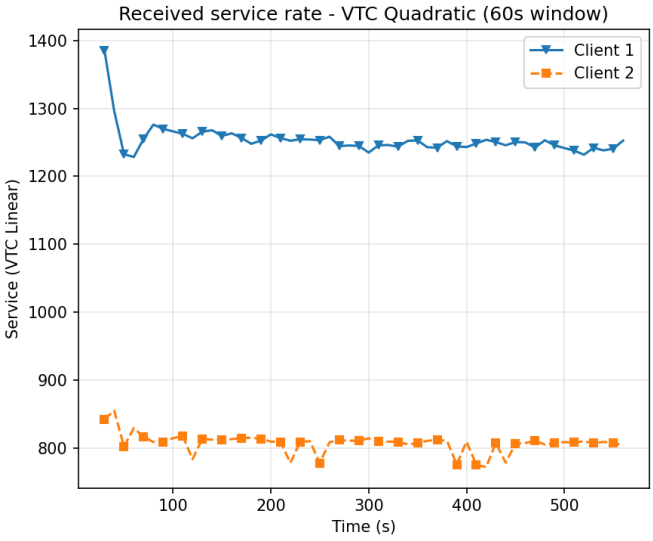


Figure 4: Service Received (using VTC Linear definition) when running Experiment 2 using the Quadratic VTC scheduler.

First, in Figure 3, we see that even with this quadratic cost function, VTC is still able to schedule new queries in a way that keeps the received service between two clients roughly equal. The larger spikes and dips in the service rate of Client 2 is largely due to the high computational and service cost of the prefill stage: on other workloads and figures you can find in our GitHub repository, using the VTC quadratic function with clients issuing smaller query sizes results in much smaller spikes and dips in the received service rate. On the other hand, since Figure 4 uses the Quadratic scheduler but graphs the received service using VTC Linear, this figure enables a much more direct comparison of the Quadratic scheduler's performance with Figure 2. Comparing Figures 1 (b), 2, and 4, we see that running the VTC Quadratic scheduler when in Experiment 2's synthetic workload results in Client 1 receiving a much closer amount of service (as defined by VTC Linear) to in Experiment 1, demonstrating how VTC Quadratic scheduling can make clients' backlogged throughput more resistant to the workloads of other clients.

## 5 Conclusion and Future Work

In summary, our contributions are the following:

- We reimplemented the VTC algorithm and modified S-LoRA to run on a more modern, higher power NVIDIA H-100 GPU, enabling us to replicate the findings from [6], including Figure 3.
- We proposed a quadratic definition of service, which adds a quadratic term based on the length of a given query.
- We show that this new definition of service, when used in the VTC scheduler, ensures that the weighted sum of tokens received by a given client becomes more resistant to the workloads of other clients when all clients are blacklogged. To the best of our knowledge, we are the first to empirically show that more accurately matching the hardware resource costs in serving leads to more resilient service for a given client, regardless of the workloads of others.

However, our findings in this paper have several limitations that could be addressed in future work. For instance, the LLM hosted in these scenarios, `llama-7b`, only supports a maximum context window of 4,096. In future work, we could explore using larger models or adding VTC to other LLM inference and serving engines to further evaluate this quadratic service definition and further optimize the weights used when calculating the amount of received service when context windows and thus input/output lengths grow larger and larger. We could also explore other service definitions, such as a tiered service definition, mimicking the tiered API cost commonly used by LLM providers by doubling the service per token if the query's input length is larger than a certain threshold. We have implemented a working version of this tiered service definition in our GitHub Repository, but evaluating this third service definition falls outside the scope of

this project.

## Software Implementation

To support the reproducability of our implementation and analysis, we have provided our implementation of VTC with both the metric gathering and experimental infrastructure required. This implemention exists on top of the S-LoRA scheduler. To view our code, please see *github.com/ullmanj/Fairness-in-LLMs-with-S-LoRA*.

## Acknowledgement

We thank professors Keith Winstein and David Mazières as well as the teaching staff of Stanford University's CS 244C for their teachings and feedback. We also thank the original authors of "Fairness in Serving Large Lanugage Models" [6] for their original contribution to this space.

## References

[1] Api pricing. In *https://openai.com/api/pricing/*.

[2] Gemini Developer API pricing | Gemini API. In *https://ai.google.dev/gemini-api/docs/pricing*.

[3] Albert Greenberg, James R Hamilton, Navendu Jain, Srikanth Kandula, Changhoon Kim, Parantap Lahiri, David A Maltz, Parveen Patel, and Sudipta Sengupta. Vl2: A scalable and flexible data center network. In *Proceedings of the ACM SIGCOMM 2009 conference on Data communication*, pages 51–62, 2009.

[4] Weixin Liang, Yaohui Zhang, Mihai Codreanu, Jiayu Wang, Hancheng Cao, and James Zou. The widespread adoption of large language model-assisted writing across society. *Patterns*, 6(12):101366, 2025.

[5] Jaime Sevilla, Lennart Heim, Anson Ho, Tamay Besiroglu, Marius Hagebeuk, and Pablo Villalobos. Compute trends across three eras of machine learning. *arXiv preprint arXiv:2202.05924*, 2022. Updated 2024; available at `https://epochai.org/ml-trends/compute-trends`.

[6] Ying Sheng, Shiyi Cao, Dacheng Li, Banghua Zhu, Zhuohan Li, Danyang Zhuo, Joseph E. Gonzalez, and Ion Stoica. Fairness in serving large language models. In *18th USENIX Symposium on Operating Systems Design and Implementation (OSDI 24)*, pages 965–988, Santa Clara, CA, July 2024. USENIX Association.

# A Extra Figures Mentioned in Paper

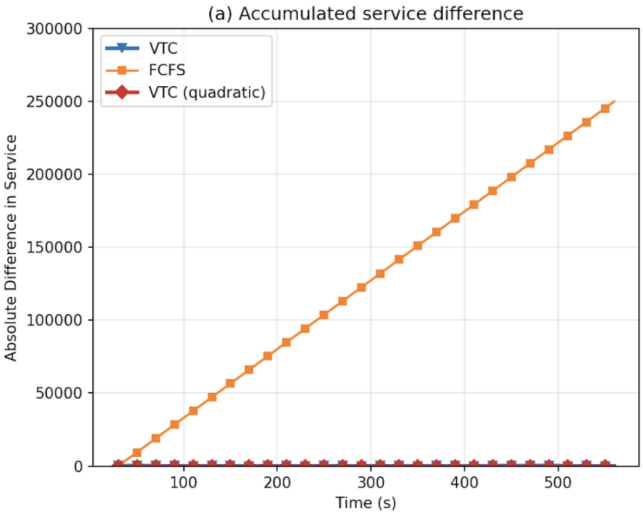


Figure 5: Figure (a) from our replication of Experiment 1 (Figure 3 from Sheng et al. [6]), including the absolute service difference measured from our VTC Quadratic scheduler.