Evaluation of Caching Strategies for Microservices

Arthur Lu, Derek Wang, Isha Atul Pardikar, Purva Gaikwad, Xuanzhe Han
University of California, Los Angeles
Los Angeles, CA, USA

ABSTRACT

Microservices are a popular architecture due to their logical separation of concerns among multiple teams of developers. However, performance and scalability remain ongoing challenges with significant research focus. One approach to improving performance and scalability is caching. In this paper, we explore advanced caching strategies and evaluate their effectiveness in accessing data within microservices. We test different eviction policies, cache topologies, and data prefetching techniques on common access patterns. Our results show that these strategies perform well on select patterns, highlighting their potential to outperform state-of-the-art solutions such as MuCache. We hope that advanced strategies can serve as a drop-in upgrade for existing microservice caches.

1 INTRODUCTION

Problem statement.

With the rise of large and complex cloud services and applications, microservice architectures have become very prevalent. While microservice architectures bring more organization and isolation to workflows and services, they introduce problems such as increased latency, network management, and scalability. Thus, many researchers in this area of research seek methods of optimizing and improving the performance of microservices.

One such area of research is in managing and optimizing communication between microservices. API calls allow services to efficiently and quickly transfer data and information to each other. However, with so many intercommunicating services, there can be network congestion and overloaded nodes that degrade performance significantly. Thus, one way of alleviating this is through caching API calls, where microservices store previously retrieved values to reduce the number of redundant API calls.

Although caching is utilized in many microservice architectures, many of these caches are simple, relying only on basic key-value retrieval, update, and invalidation. It is possible that proprietary microservices utilize state-of-the-art API call caching, but publicly available information on their caching strategies is hard to find. Additionally, most microservice caching papers are application-based, focused on regional/large-scale caching, or compare very few caching strategies together.

State of the art.

The state-of-the-art for microservice API caching is MuCache, which proposes a system framework that automatically provides microservice applications with inter-service caches that improve performance [8]. The paper indicates that their framework results in lower latency, lower resource usage, and faster invalidation. However, the caching strategies utilized are simple, with MuCache using a simple key-value cache layout, LRU eviction policy, and no prefetching. In addition, the authors do not delve into the impact of different caching strategies.

Both Azure and AWS documentation also discuss caching for microservices [4] [3]. Similar to MuCache, the caching strategies that are discussed are simple key-value stores using Redis or Memcached. They also do not feature different architectures or prefetching and only discuss simple time-based eviction policies.

Key challenges.

One key challenge in researching microservice caching is determining how to create a realistic testing environment for each caching strategy. Microservices can be large-scale and diverse, making it difficult to simulate the complex interplay between services that arise in these deployments. It is also important to test various cache strategies without potentially introducing too many additional variables to account for.

Another challenge is how to set up testing and evaluation metrics to accurately compare and contrast different caching strategies. Each strategy has its pros and cons as well as an optimal environment to operate within. We choose a range of measurements which best represent a cache's performance and compare advanced cache strategies against the state-of-the-art.

Key insights/solutions.

Our goal is to analyze the impact and effectiveness of various caching strategies for API calls within microservice architectures. By comparing different caching techniques and eviction policies, the results inform developers with actionable insights to optimize caching decisions and select the most suitable eviction strategies based on empirical evaluation.

When determining which caching strategies to analyze, we focus on promising strategies that are expected to reduce latency in microservice systems. We selected four strategies relating to eviction strategies, multi-tiered caching, and prefetching. These strategies have promising results in other applications/systems, making them good candidates for caching API calls.

Evaluation.

To evaluate each caching strategy, each cache strategy is tested on different workloads in a mock social media environment. This is not a complete microservice environment but is a small fragment of two services that might be in a microservice application environment. We assume that caching strategies between two services can transfer to other service pairs as well. This approach can save resources and quickly test our caching strategies.

As mentioned, a difficult task is to decide on the metrics to compare the performance of each caching strategy. Each strategy performs better in some metrics than others. In this project, we measure the performance of our strategies via hit ratio, average response time, average cache hit/miss response time, cache throughput, and real throughput.

Contributions.

With this paper, we conducted a synthetic analysis on the impact and effectiveness of various caching strategies for API calls within microservice architectures. We discovered caching strategies that work well in certain workloads in a microservice environment. We encourage more research into the impact of caching strategies within microservices.

2 OVERVIEW

As mentioned, simulating an entire microservice environment is costly and hard to analyze. Thus, we assume that if a caching strategy improves the performance of the API calls between two individual microservices, this performance boost should reflect on the entire environment. This has been demonstrated in recent research such as MuCache [8]. We can make this assumption because API calls are dependent only on the two services. For example, if a specific strategy is effective for a high read workload between two nodes, then it should also apply to other node pairs with high read workloads. In a realistic environment, this assumption is challenged by the fact that microservices are dynamic and complex, which does not guarantee a static workload distribution between nodes. However, this dynamic nature can be simulated and can give us accurate insights into how these caching strategies might perform in a real environment. The testing environment is small simulation that mimics the communication between two microservice nodes. With this testing environment, each cache strategy can be tested effectively on different workload patterns.

Although caching has been utilized in many microservices, there are still improvements that can be made even to the state-of-the-art microservices. For example, when Meta characterizes the workload of their caching system, MemCache, they discuss the distribution of cache misses in their systems [5]. From their findings, they found that a majority of their cache misses are eviction and compulsory misses. In fact, for some pools in Meta, 99% of the cache misses are compulsory misses. This highlights a need to improve cache eviction policies and prefetching within microservices. For our caching strategies, we decided to focus on prefetching, tiered caching, and sieve, which can help reduce these types of cache misses.

3 METHODOLOGY

We implemented a simple abstract class for the cache, providing the basic functions shown in figure 1. We use this abstraction to quickly implement different caches and test them in the simulated environment without any concern. We then implement each of our cache strategies and a baseline cache on top of the abstract class. We selected the following caching strategies: **sieve**, **speculative prefetching**, **tiered caching**, and **read-after-write prefetching**. We implement three baseline comparisons to evaluate our strategies: a no-cache baseline, a state-of-the-art cache utilizing LRU eviction without prefetching, and an ideal cache. The ideal cache serves as a hypothetical upper bound on caching performance. Each caching strategy is different and has unique designs to improve its performance and hit rate.

```
get(key) -> return item value given key or None
put(key, value) -> insert item to cache
invalidate(key) -> mark item with key as invalid
```

Figure 1: Cache Abstract Class Methods

Baselines

Our baseline cache model is a straightforward key-value cache that uses the least recently used (LRU) eviction strategy. This method is widely used, including in MuCache, and is known to work well. To better understand how our caching strategies compare, we also introduce two extreme cases: ideal caching and no caching. In an ideal cache, every request results in a hit, while in a no-cache scenario, every request is a miss. These two extremes help set the performance boundaries, giving us a clear reference point when evaluating our caching strategies.

SIEVE

SIEVE is a simple but efficient cache eviction algorithm that improves cache management by using a single queue along with a "hand" pointer for evictions [6]. It remembers if an object has been accessed using a single "visited" bit. Mostly used items remain in the cache, while the least used items are evicted when the hand traverses from the tail to the head of the queue. This design reduces the computational overhead and increases the throughput.

In an Instagram-like high-read workload environment, user profile data, follower lists, and activity feeds generate many API requests. Traditional caching methods may impose excessive complexity and computational overhead. SIEVE's minimalist design and low-overhead eviction policy allow for more efficient use of the cache with frequent access user data still being easily accessible and preventing redundant unnecessary API calls.

By using SIEVE caching in the Instagram user model, we expect reduced cache miss ratios, less API latency, and quicker response times. SIEVE's performance and low-overhead architecture can scale better, allowing Instagram to handle large-scale user interactions more effectively. Furthermore, unlike traditional LRU-based caches, SIEVE reduces costly cache promotions, which optimizes further for performance.

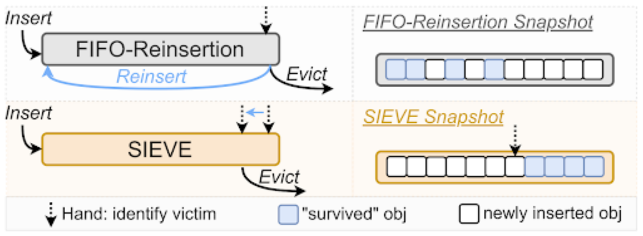


Figure 2: Sieve Eviction Algorithm

Speculative Prefetching

Prefetching represents caching strategies that utilize the data context to improve caching based on locality. When data is fetched and put in the cache, other data that is likely to be accessed next is preemptively fetched into the cache.

In our implementation, when a user profile is fetched from the database, several random friends of that user are also fetched in anticipation of the possibility of them being accessed shortly after. For evaluation, the number of friends' profiles that are fetched is 10% of the cache size, as we use the number of items instead of size in bytes as the cache size limit. This number can be adjusted by the user, but we found that 10% worked well for our particular environment.

LRU is used as the eviction policy. If the cache is full, prefetching is still performed and items will be evicted to accommodate the prefetched items. There are no other special invalidation methods implemented. This type of contextual-locality caching can be done for other systems using developer knowledge, and can either be hard-coded in or learned using some machine learning techniques.

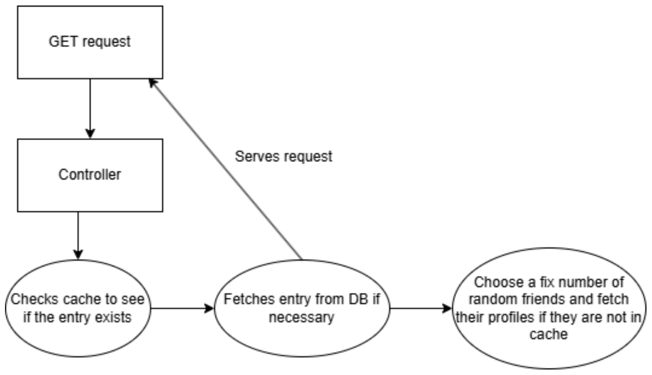

GET request
Controller
Serves request
Checks cache to see if the entry exists
Fetches entry from DB if necessary
Choose a fix number of random friends and fetch their profiles if they are not in cache

Figure 3: Speculative Prefetch Workflow

Tiered Cache

We extend the baseline model with multiple-tier caching. Tiered caching uses multiple layered tiers of caches to effectively create a large cache structure with short response times. Each descending layer features more cache space but higher cache latency. Our implementation uses a 2 tier cache design using memory and disk. Like the baseline, the L1 cache stores entries in a memory hash table. Then, we also add an in-memory lookup table for the L2 cache which stores entries on the file system. We use LRU for eviction on both tiers and did not add any special invalidation or prefetching strategies. We reduced the L1 cache size by half compared to other strategies to allocate space for the L2 lookup table.

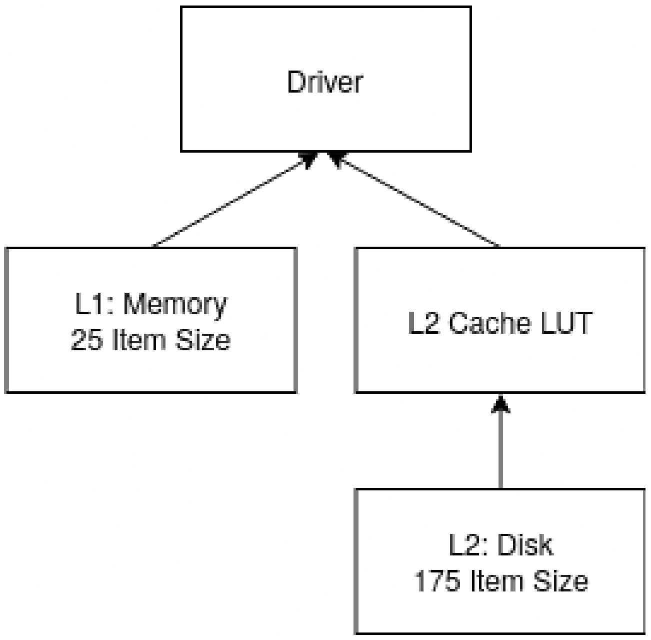


Figure 4: Tiered Cache Architecture

Read-After-Write

Read-After-Write makes the assumption that when we write data, that new data may be read soon after. The basic strategy is to update the data in the cache as well when conducting write operations. This way, the data can be quickly accessed when it's being read after being written. While this general strategy is applicable to any arbitrary application, our implementation is not as robust due to the simple design of our API simulation and cache. When the baseline cache receives a write request, the existing old data would be invalidated and removed from the cache. When the Read-After-Write intercepts a write request, it instead fetches the user data and replaces the invalid data as seen in Figure 5. While this method is roundabout, we believe that the performance difference from the actual method is negligible and still demonstrates its effectiveness.

Even though the strategy is simple, it requires the developer to have some knowledge/context about the workloads between two nodes to satisfy the assumption mentioned before. Some workloads might be dynamic or hard to analyze, making this strategy difficult to apply for every situation. In addition, we are only testing with single data write requests. If we write multiple data, there will be a latency/computation penalty that scales as we interact with more data. This could impact the effectiveness of the strategy, but will not be addressed for this project. However, this can be studied in future research.

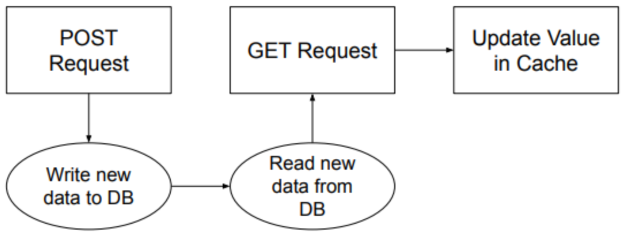


Figure 5: Read-After-Write Implementation Workflow

4 EVALUATION

We evaluate our caching strategies within a simulated microservice environment, measuring their performance under different workloads. Our evaluation is structured to answer the following key questions:

- How do different caching strategies impact performance in microservice environments?
- Under what conditions does each caching strategy provide the best benefits?
- What are the trade-offs between caching techniques in terms of latency, throughput, and cache efficiency?

To answer these questions, we describe our testing infrastructure, evaluation methodology, benchmarks, workloads, and results.

4.1 Testing Infrastructure

To analyze and compare our strategies, we designed a simulated microservice environment. Instead of deploying a full-scale distributed system, we assume that caching performance observed in API calls between two microservice nodes can generalize to other microservice pairs. This allows us to conduct controlled and repeatable experiments efficiently.

Microservice Simulation

We built a small-scale mock social media platform where users have friends, followers, and posts. The platform is composed of two nodes, and a TinyDB database, a lightweight NoSQL database, that stores user profiles. To simulate the interaction between two nodes, we utilized FastAPI, which allows us to run a local server to send GET and PULL requests. Our FastAPI-based microservice sends GET and POST requests to retrieve or update user profiles, while another node interacts directly with the database to serve the requests. This mimics the read-and-write API calls between microservice nodes. The caching layer is implemented between the API endpoints and the underlying database. Data is stored in TinyDB, a lightweight NoSQL database.

Data Generation

For our experiment, we populated our database with dummy user profiles. Each user profile consists of a user ID, a name consisting of first and last names, a follower count that is just a random integer, a short sentence (around 10 words) bio and post, and a list of 1 to 50 user IDs representing friends. Each profile is around 500 bytes.

These profiles are generated using DeepSeek R1 with the Huggingface endpoint. For reference, the model was set for text generation, with a temperature of 1.0, top k of 60, and top p of 0.9 to control the randomness of the output. The max token count is set to 150. The model is invoked to generate one profile at a time, with the output filtered using regular expression. If the output is syntactically correct, it is put in the database. 1002 profiles are generated this way

We acknowledge that this dummy dataset is far from perfect. There are patterns with real user profiles, like having similar friends if they belong to the same friend group or having patterns in their posts and bios. However, with the limited time we had, we were unable to located a good dummy dataset that could be converted to serve our framework. With the use of LLM, we were able to generate enough random data quickly with enough randomness to simulate real-world data.

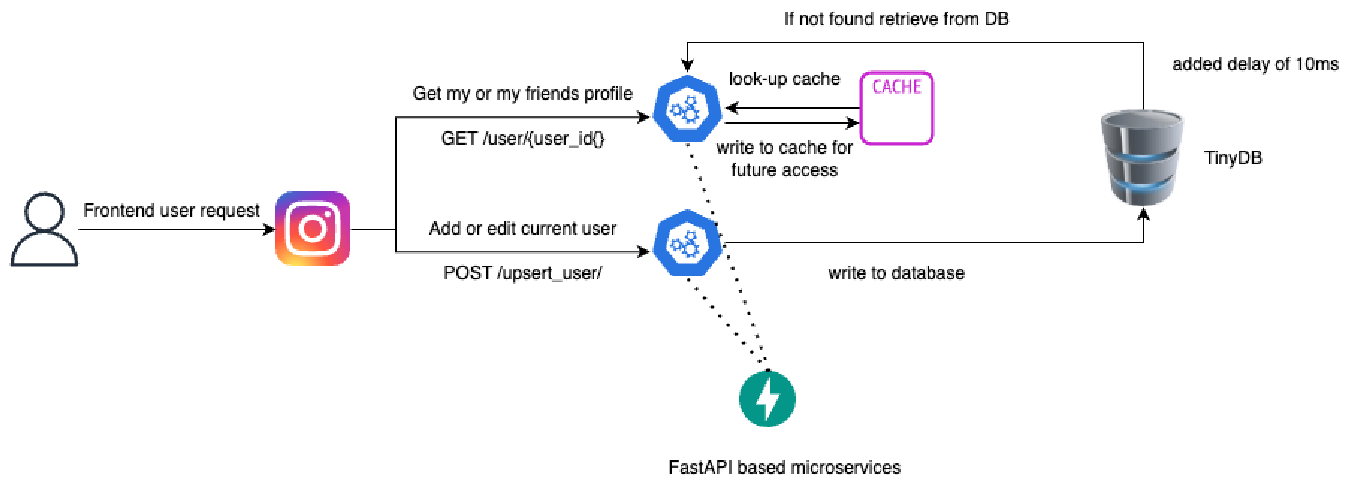


Figure 6: System Architecture

Artificial Network Latency

Since both FastAPI and TinyDB run in memory on the same machine, we implemented an artificial 10 ms delay between them. This reflects the roughly 10-12 ms observed tail latencies from research in characterizing microservice performance [1]. By adding 10 ms, the total end-to-end latency more closely reflects real world deployments.

Hardware Setup

All experiments were conducted on Ubuntu Server 24.04 LXC container on a Ryzen 3600X machine with added network delays for database communication.

4.2 Evaluation Metrics

Each caching strategy optimizes different performance metrics. There are also many ways to measure cache performance and we choose the below metrics based on research [2]. Thus, we measure the following metrics:

- **Hit Ratio**: Fraction of requests served by the cache instead of the database. Higher hit ratios indicate better caching performance.
- **Average Response Time**: End-to-end latency, including both cache hits and misses.
- **Cache Hit Latency**: Time taken to retrieve a cached item (lower is better).
- **Cache Miss Latency**: Time taken when a cache miss occurs, including the database fetch.
- **Cache Throughput**: Requests per second served by the cache.
- **Real Throughput**: Total requests per second processed by the system.

4.3 Workloads and Benchmarks

We test our caching strategies across a variety of realistic workloads, inspired by common access patterns in social media and networked applications.

Simulated Workloads We evaluate each caching strategy on 10,000 requests per workload, each simulating API requests between microservices. Since there can be many patterns of requests, we model 3 read heavy workloads and 2 write heavy workloads, which are common patterns in microservice applications [7]. We also add include patterns which reflect data associativity by leveraging user associations.

- **Random Read-Only (100% Reads)**: Emulates analytics or web crawling where reads are purely random.
- **Read-Heavy (80% Reads, 20% Writes)**: Simulates a microservice handling frequent profile views with occasional updates.
- **Write-Heavy (80% Writes, 20% Reads)**: Represents applications with frequent updates (e.g., status changes).
- **Frequent Users (Top 10 Users Accessed 70% of Time)**: Models scenarios where a subset of users (e.g., influencers) receive disproportionate traffic.
- **Frequent After Write (R after W)**: A user who just updated their profile or posted content is highly likely to be accessed soon.
- **Friend-Based Access (Friend Read-Only)**: Users access a profile and then, with a high probability, access the user's friend's profile.

Friend Read-Only Variations. For friend-based access, we experiment with different probabilities of accessing a friend after a user profile lookup. We test 25%, 50%, and 75% probabilities.

4.4 Results and Analysis

We analyze the experimental results from our evaluation across different caching strategies. The raw results are included in

Cache Miss Ratio.

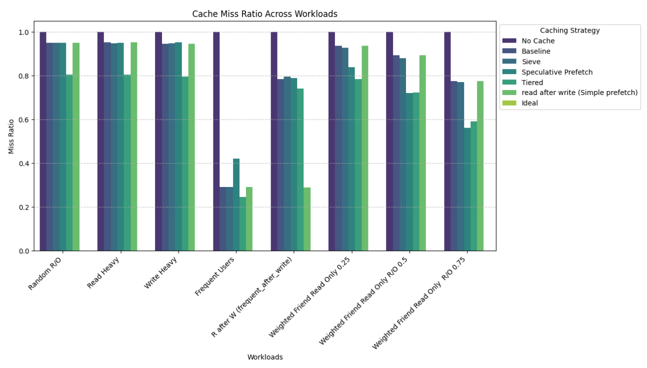


Figure 7: Cache Miss Ratio Across Workloads

Across read heavy workloads, Tiered cache and Speculative Prefetching performed well with the lowest miss rate. Note that the ideal cache is not shown because its hit rate is always 0. Sieve performed about as well as the baseline in all tests. On write heavy workloads, Speculative Prefetching and Read-After-Write performed well on the friends based associative workload and read after write workload, respectively.

Throughput Comparison.

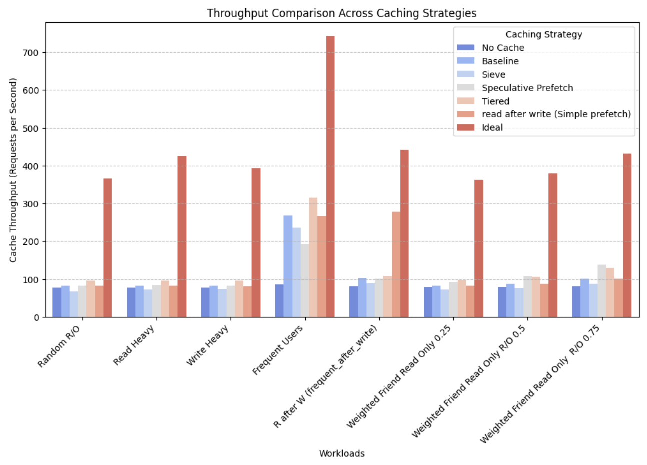


Figure 8: Throughput Comparison Across Caching Strategies

The throughput mostly reflected the miss rate of each cache and shows that Tiered and Speculative Prefetching performed well on most read heavy tasks. Read-After-Write also had high throughput on the read after write test. However, SIEVE perfomed significantly worse than the baseline. Also note that each strategy is still far from the ideal cache throughput, indicating more room for improvement. Finally, write-heavy workloads showed lower throughput, indicating that traditional caches struggle may frequent updates.

Cache Miss Latency.

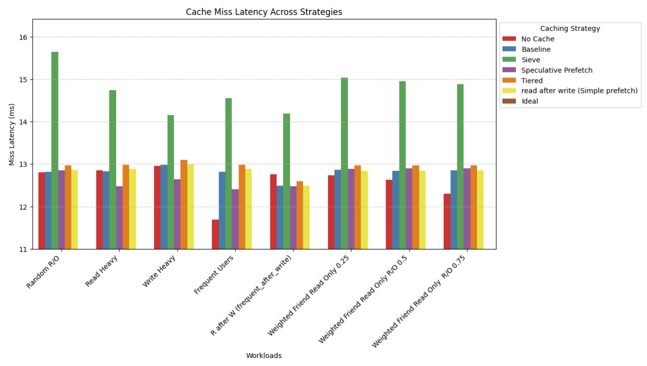


Figure 9: Cache Miss Latency Across Strategies

This test shows the cache's latency overhead on a cache miss. Ideally, there would be minimal overhead. Most strategies had similar average cache miss latency compared to the baseline, which is expected. However, SIEVE had significantly more latency on cache misses, which explains its poor throughput performance. We discuss reasons for this in 4.6. Ideal cache has no miss latency.

Cache Hit Latency.

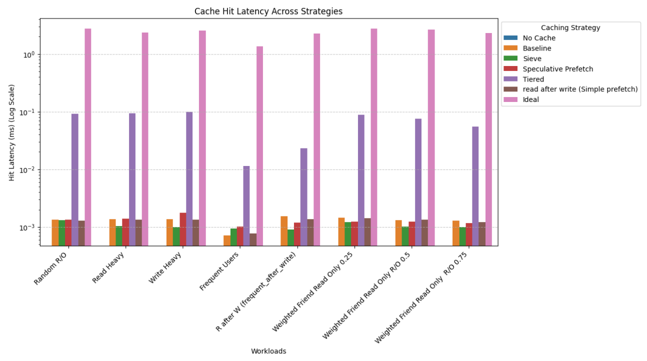
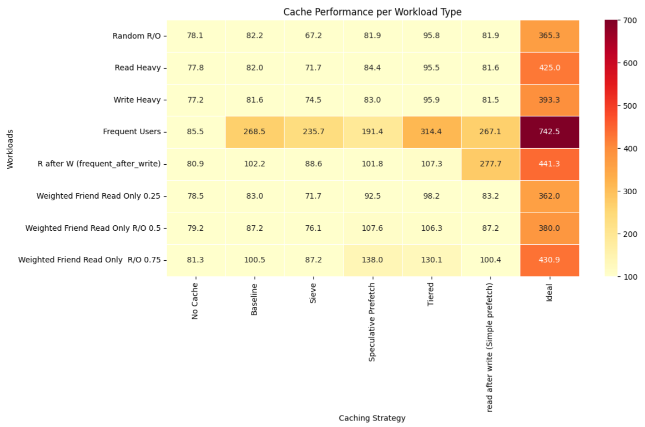


Figure 10: Cache Hit Latency Across Strategies

This test measures the cache's latency on a cache hit. The results show that most caches were able to respond in a few microseconds, which reflects the latency of system memory.

However, the Tiered cache had significantly higher latency, which is likely due to retrieving cache items from the file system. The ideal cache had the highest latency of several milliseconds simply because the database was used as the cache, and this is reflected in the latency. No cache does not have hit latency.

Cache Performance Per Workload Type.



Workloads	No Cache	Baseline	Sieve	Speculative Prefetch	Tiered	read after write (Simple prefetch)	Ideal
Random R/O	78.1	82.2	67.2	81.9	95.8	81.9	365.3
Read Heavy	77.8	82.0	71.7	84.4	95.5	81.6	425.0
Write Heavy	77.2	81.6	74.5	83.0	95.9	81.5	393.3
Frequent Users	85.5	268.5	235.7	191.4	314.4	267.1	742.5
R after W (frequent_after_write)	80.9	102.2	88.6	101.8	107.3	277.7	441.3
Weighted Friend Read Only 0.25	78.5	83.0	71.7	92.5	98.2	83.2	362.0
Weighted Friend Read Only R/O 0.5	79.2	87.2	76.1	107.6	106.3	87.2	380.0
Weighted Friend Read Only R/O 0.75	81.3	100.5	87.2	138.0	130.1	100.4	430.9

Figure 11: Cache Performance Across Strategies

We again show the throughput for each cache on each workload. The heatmap shows which caches are best suited for each workload. Tiered cache performed well across many workloads, and both prefetching caches performed well on their respective workloads. In general, caches perform best under workloads with strong locality patterns, and did not perform well without power-law assumptions.

4.5 Insights for Developers

Based on these results, we provide the following insights for microservice developers:

- **For Read-Heavy Workloads:** Speculative Prefetching and Tiered Caching are the best choices.
- **For Write-Heavy Workloads:** None of the tested caches performed well. More research is needed into write-optimized caches.
- **For Social Media and API Calls:** Read-After-Write Caching and Prefetching perform the best.
- **For Large-Scale Applications:** Tiered Caching effectively balances storage size and latency.
- **For Friend-Based or Power-Law Workloads:** Prefetching and Read-After-Write outperform traditional LRU caches.
- Utilizing extra disk space could be a useful approach to increasing cache performance. If a microservice instance has access to some extra unused disk space which is low latency, developers should utilize this. This result also highlights a possible justification for cloud providers to include or sell some disk space for this use as well. If storage cost is less than memory cost, cloud providers may save money if microservice caches switch to a multi-tiered model which leverages memory and disk together.

4.6 Limitations and Future Work

- **Simulation vs. Real Deployments**: Our experiments were conducted in a controlled simulated environment. Future work should test these caching strategies in real-world microservice deployments such as *DeathStarBench*. Additionally, future work can modify the MuCache architecture directly and test improvements on its caching strategy.
- **Tiered Caching I/O Considerations**: In our experiments, the multi-tiered cache performed well due to its balance of response time and cache size. They showed that an increase to cache size at the cost of latency can still result in improved throughput and average end-to-end latency. However, this is only the case if both the L1 and L2 caches have significantly lower latency than the network. As a result, our results may not reflect real microservice deployments where disk space may often be on a network attached storage. It remains to be seen whether this type of cache will still perform well, although it seems unlikely since the network latency would not be bypassed even on a cache hit.
- **Impact of Workload Distribution**: Our study suggests that power-law workloads (e.g., Frequent Users) are fundamental for effective caching. More research is needed into realistic microservice access patterns. Future experiments could experiment with traces obtained from real-world APIs (e.g., Twitter API).
- **Dynamic Cache Strategy Switching**: In real microservice environments, workloads are sporadic and dynamic, making these static strategies suboptimal. Future research can study dynamic caches that change policies based on analysis of a given workload.

5 CONCLUSION

Our analysis of caching strategies in microservices provides a foundation for future research and practical caching guidelines for developers. We show the impact of caching strategies on various microservice workloads. By understanding the trade-offs between eviction, prefetching, and tiered caching, developers can design optimized, workload-specific caching strategies for their applications. We show that each strategy works well for certain workloads, with prefetching and multi-tiered cache performing the best all-round. There is also room to improve on our results, especially with improving our testing methodology. Future experiments should attempt to utilize existing benchmarks and explore more microservice-centric caching strategies to further analyze the impact of caching strategies in microservice environments. Ultimately, we hope this project can inspire future research in this area.

REFERENCES

[1] Gan, Y., Zhang, Y., Cheng, D., Shetty, A., Rathi, P., Katarki, N., Bruno, A., Hu, J., Ritchken, B., Jackson, B., Hu, K., Pancholi, M., He, Y., Clancy, B., Colen, C., Wen, F., Leung, C., Wang, S., Zaruvinsky, L., Espinosa, M., Lin, R., Liu, Z., Padilla, J., and Delimitrou, C. An open-source benchmark suite for microservices and their hardware-software implications for cloud & edge systems. In *Proceedings of the Twenty-Fourth International Conference on Architectural Support for Programming Languages and Operating Systems* (New York, NY, USA, 2019), ASPLOS '19, Association for Computing Machinery, p. 3–18.

[2] Mertz, J., and Nunes, I. Understanding application-level caching in web applications: A comprehensive introduction and survey of state-of-the-art approaches. *ACM Comput. Surv. 50*, 6 (Nov. 2017).

[3] Microsoft. Caching guidance.

[4] Saleem, I., Nargund, P., and Buonora, P. Data caching across microservices in a serverless architecture, 2021.

[5] Xu, Y., Frachtenberg, E., Jiang, S., and Paleczny, M. Characterizing facebook's memcached workload. *IEEE Internet Computing 18*, 2 (2014), 41–49.

[6] Yang, J., Zhang, Y., Yue, Y., Vigfusson, Y., and Vinayak, R. Sieve: Cache eviction can be simple, effective, and scalable. *USENIX* (Jun 2024).

[7] Yuehai Xu, Eitan Frachtenberg, S. J. Building a high-performance key-value cache as an energy-efficient appliance. *IFIP* (2014).

[8] Zhang, H., Kallas, K., Pavlatos, S., Alur, R., Angel, S., and Liu, V. MuCache: A general framework for caching in microservice graphs. In *21st USENIX Symposium on Networked Systems Design and Implementation (NSDI 24)* (Santa Clara, CA, Apr. 2024), USENIX Association, pp. 221–238.