Cache Me If You Can: Taming the Caching Complexity of Microservice Call Graphs

Abhishek Tiwari

Citation: *A. Tiwari*, "Cache Me If You Can: Taming the Caching Complexity of Microservice Call Graphs", Abhishek Tiwari, 2024.
doi:10.59350/cq9aw-a9821

Published on: December 17, 2024

As microservices architectures have become increasingly common in modern software systems, they have brought benefits and challenges. One of the most pressing challenges has been maintaining performance at scale while dealing with complex service dependencies and network communication overhead. Today, we want to explore MuCache, a caching framework recently presented by Zhang et al. at USENIX NSDI 2024 that tackles this challenge head-on by providing automatic and coherent caching for microservices call graphs (see [1]).

The Microservice Caching Dilemma

Before exploring MuCache, it is important to grasp why caching in a microservices architecture is essential and complex. In a monolithic application, the function calls among components are simple and quick. However, in a microservices architecture, these same interactions become a network of calls, introducing significant latency and resource overhead, which is why companies like Alibaba have found that without effective caching, their service call graphs can reach depths of more than 40 calls. With proper caching, they can reduce this to just three calls for many requests.

Cache Coherence

The main challenge is ensuring consistency among caches. Unlike traditional cache coherence, which deals with individual read/write operations, microservices depend on complex webs of downstream service calls and state changes. When data is updated in a service's database, all caches holding that data need to be updated or invalidated. This coordination is complex, especially in a distributed microservice environment.

Consider the complexity of managing a social network's home timeline - a feature we are all familiar with from platforms like Twitter. The caching challenge here runs deeper than it might first appear. A user's timeline is not just a list of posts but a collection of different data points and business rules working together. Keeping the cache updated is particularly challenging when we consider all the reasons that might require refreshing the cached timeline.

Consider all the moving parts: when someone you follow adds a new post, another user changes their privacy settings, or a post disappears from the timeline. Additionally, the ranking algorithm may change to highlight more specific types of content. This situation is complicated because many of these changes come from services that do not directly interact with the timeline creation service. These changes affect the timeline's accuracy through complicated connections.

Even when some changes affect the final timeline, parts of the underlying data may still be valid and can be stored. For example, if a user updates their profile picture, we do not need to redo their post

content or invalidate cached follower relationships. Using this detailed approach to caching can help a system not only function but also perform well as it scales.

This rewrite maintains the technical depth while making it more conversational and relatable, using familiar examples to illustrate the complex concepts. It also emphasises the nuanced nature of the caching challenge while making it more accessible to readers.

Limited State-of-the-Art

When developers face challenges with caching in microservices, they often choose one of three less-than-ideal solutions.

First, they may decide to create their own custom coherence protocols tailored to their application. While this approach gives them control, it can result in fragile solutions with many edge cases and a struggle to adapt as systems change.

Focusing only on caching at the backend storage layer is a common approach. However, this method misses an important opportunity: it doesn't reduce the long call chains that requests have to go through before reaching the backend. By the time a request arrives at the backend, we have already dealt with the delays caused by multiple network hops and service calls.

Finally, many teams wave the white flag on strict consistency guarantees and implement basic time-to-live (TTL) caching mechanisms. While this approach is simple to implement, it essentially trades correctness for performance. Users might see stale data, and developers are left with the unenviable task of tuning TTL values - too short and one loses the benefits of caching; too long and one risks serving outdated information.

Current service mesh frameworks like Dapr, Envoy, and Istio provide robust solutions for service discovery, load balancing, and fault tolerance, but they've largely avoided inter-service caching. The complexity of maintaining cache coherence across a dynamic service graph has made this a particularly challenging problem to solve in general.

Key Requirements

We must carefully balance several critical requirements when designing a caching solution for microservice architectures. First and foremost stands correctness - any caching system we implement must act as a transparent performance optimisation layer. It should never introduce new behaviours or alter the application's existing semantics. Think of it like a guitarist's effects pedal - it should enhance the sound without changing the fundamental melody.

It is also essential to have non-blocking operations that incur little overhead. In the high-stakes world of distributed systems, we cannot afford to introduce delays on the critical path of request processing. A caching layer should function with the subtlety of a shadow, providing performance advantages without incurring substantial expenses. Any synchronisation or coordination should occur in the background, separate from the primary request flow.

The third vital requirement addresses the dynamic nature of modern microservice architectures. Unlike traditional monolithic applications where the component relationships are fixed at compile time, microservice call graphs are often determined at runtime. A single request might take different paths through the service mesh depending on user context, system load, feature flags, or business rules. A caching solution must gracefully handle this dynamism, adapting to changing call patterns without requiring predefined knowledge of the service topology.

Enter MuCache

MuCache simplifies caching management in microservice architectures while ensuring strong consistency. MuCache pushes cache management off the critical path of request processing while maintaining correctness by carefully tracking dependencies and invalidations.

The system works by introducing three key components (see following illustration):

1. Wrappers that intercept service invocations and database operations
2. Cache managers that track dependencies and handle invalidations
3. The actual cache storage (which can be any standard cache like Redis)

Wrappers (W): Special functions in the sidecar (see [2]) that intercept service invocations and database operations. They check the cache before invocation and return the result if there is a cache hit. Wrappers also supply context information to the cache manager, such as input arguments, request start and end timestamps, and the keys that the request reads and writes.

Cache Managers (CMs): Standalone processes that receive information from the wrappers and decide when to save or invalidate the cache. Each service has its cache manager collocated with it. The wrappers do not wait for any responses from the cache manager, ensuring that the client is never blocked.

Datastore: The actual cache used, such as Redis or Memcached. For now, MuCache only supports linearisable data stores as a cache backend.

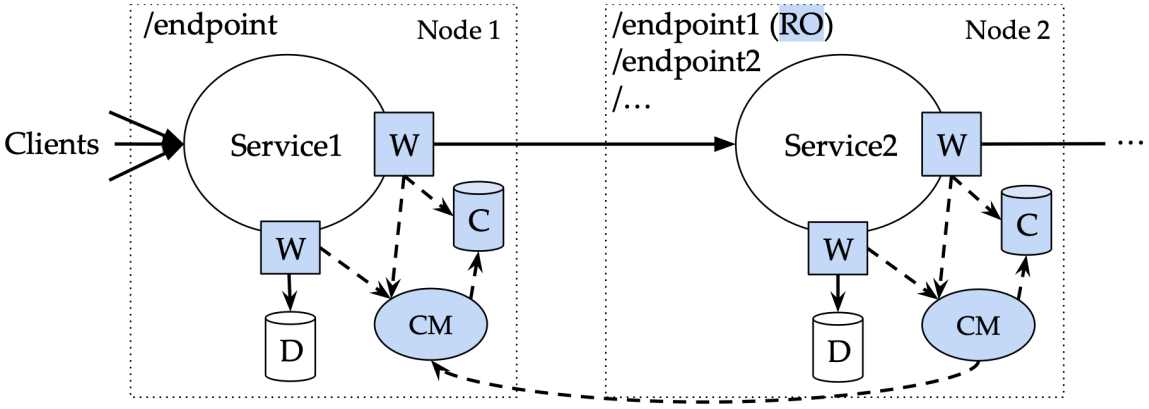

Figure 1: MuCache's Architecture. (C) denote caches, (CM) cache managers, (W) wrappers, and (D) the data stores. Wrappers are interceptor functions in the sidecar of each service. Solid arrows denote baseline communication, while dashed arrows and blue components denote additions by our system. RO means read-only. Image credits Zhang et al.

What makes MuCache particularly interesting is its approach to consistency. Instead of maintaining strict consistency across all services (which would require expensive synchronisation), it introduces a novel correctness condition based on client-observed behaviour. This allows for reordering operations between independent clients while maintaining consistent behaviour from each client's perspective.

The Protocol

At its core, MuCache operates on a simple principle: identify which service endpoints are read-only and intelligently cache their responses. Developers start by declaring which methods in their service APIs never modify the system state. For services using REST, MuCache can simplify this process by automatically treating GET endpoints as read-only, a common convention in REST architectures.

MuCache's lazy-invalidation cache coherence protocol is explicitly designed for the dynamic nature of microservices call graphs. The Mucache protocol brings two important features to the table. First, it eliminates the need to block operations during cache access. Whether a request hits or misses the cache, the request keeps moving forward without waiting for other requests to complete. This design choice optimises the happy path of cache hits and ensures that cache misses incur minimal, predictable overhead.

The second is how MuCache maintains consistency. The protocol is not just fast - it is provably correct, offering a powerful guarantee: any behaviour you observe in a system with MuCache enabled

could have occurred in the original system without caching. Correctness ensures developers can confidently add caching without worrying about introducing new, unexpected behaviours into their applications.

The Implementation Details

MuCache's implementation is particularly elegant in handling the complexity of the microservices call graph. When a service makes a read-only call to another service, the wrapper first checks if the result is cached. If there is a cache hit, it returns immediately. If not, it forwards the call while tracking all keys read during the request's execution.

The cache manager maintains two critical pieces of state: a saved map that tracks which upstream services have cached which responses and a history of calls and invalidations. When a write occurs, the cache manager uses this information to determine which cached entries need to be invalidated.

This design is particularly clever in handling the "diamond pattern" problem - where a service might access the same downstream service through multiple paths. MuCache tracks the set of services visited during request processing and avoids using cached entries if they depend on a service already visited in the current request path. This prevents consistency violations that could occur from accessing stale data through different paths.

The MuCache prototype is written in Go and has about 2,000 lines of code. The authors have shared it on Github.

Performance and Real-World Impact

The performance enhancements exhibited by MuCache are interesting. During tests with practical applications, it demonstrated:

- Up to 2.5x reduction in median latency
- Up to 1.8x reduction in tail latency
- Up to 60% increase in throughput
- Only 13% CPU overhead on average

What is particularly noteworthy is how MuCache performs compared to backend-only caching and TTL-based approaches. It consistently outperforms backend-only caching while staying close to the theoretical maximum performance of TTL∞ (infinite TTL) approaches without sacrificing consistency guarantees.

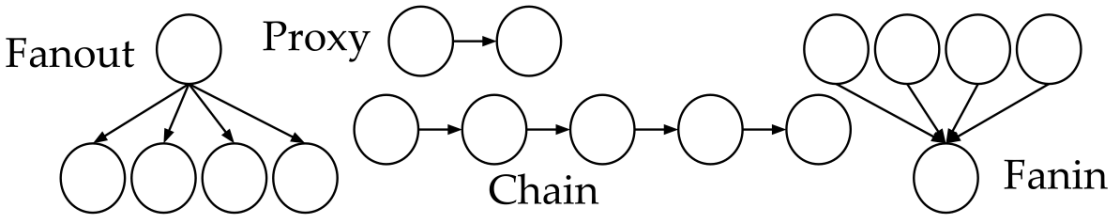

Figure 2: MuCache tested four synthetic patterns: proxy (simple two-tier), chain (sequential), fan-out (one-to-many), and fan-in (many-to-one). While proxy established baseline overhead metrics, chain patterns showed 2.6-3.1x lower median latency, fan-out improved tail latency by 1.6x, and fan-in achieved 1.75x higher throughput.

The system also shows excellent scalability characteristics. MuCache maintained its performance advantages when testing with sharded services while scaling linearly with the number of shards. This is achieved because cache managers of different shards only communicate invalidations in the background, keeping the critical path free from cross-shard synchronisation.

Practical Considerations and Limitations

MuCache significantly improves caching, but knowing its limitations and requirements is crucial.

Linearisable Datastores

MuCache relies on linearisable data stores, which might not be available in all environments, thus limiting MuCache's ability to function with many widely-used data stores. For example, by default, numerous NoSQL databases such as Cassandra and MongoDB provide eventual consistency in contrast to linearizability. Even some SQL databases, when configured for high availability or geographic distribution, may sacrifice linearisability for better performance or partition tolerance.

Sidecar Overhead

The current implementation is built on top of Dapr, which makes integration straightforward for services already using Dapr but might require additional work for services using other communication mechanisms. The reliance on service mesh sidecars presents an interesting tension in MuCache's design. As Zhu et al. (see [3]) revealed significant performance penalties associated with service mesh

sidecars - with sobering numbers like 269% higher latency and 163% increased CPU utilisation - MuCache's actual performance data tells a nuanced story about this trade-off.

MuCache's implementation shows that the overhead can be well worth the benefits in many scenarios. The system reports only a 13% increase in CPU usage while delivering up to a 2.5x reduction in median latency and a 60% increase in throughput, which suggests that the caching benefits can substantially outweigh the sidecar overhead, particularly for applications with expensive service-to-service calls or complex call graphs.

However, this raises important considerations. In systems where service-to-service calls are relatively lightweight or where the call graph is shallow, the sidecar overhead might eat into the potential benefits of caching. The decision to implement MuCache should, therefore, consider the specific characteristics of the application—particularly the cost of service calls relative to the sidecar overhead.

GC Overhead

Another practical consideration is the memory overhead of tracking dependencies. While the paper shows minimal overhead (less than 0.4% of cache size on average), systems with extremely high request rates or complex dependency graphs need to tune the garbage collection parameters to maintain reasonable memory usage.

Cache Poisoning

Lastly, introducing any caching layer in a distributed system creates potential security vulnerabilities, and cache poisoning represents a particularly concerning threat to inter-service caching systems like MuCache. In traditional systems, cache poisoning typically affects a single service or endpoint. However, due to the interconnected nature of services, the implications could be more severe in a microservices architecture with inter-service caching.

MuCache has a higher risk because it reuses cached results across different request paths. If someone gains access to the service, they could put harmful data into the cache. This harmful data could then be served to other services without being rechecked. This is concerning since MuCache is designed for speed and does not perform real-time checks during critical processes.

Future Direction

MuCache approach to automatic cache coherence could become a standard feature of service mesh implementations, similar to how automatic retry logic and circuit breakers are today.

The system's design also provides valuable insights for the broader field of distributed systems. The approach to maintaining consistency through carefully tracking dependencies while avoiding synchronisation on the critical path could be applied to other distributed caching problems beyond microservices.

For practitioners, MuCache offers a path to achieving the performance benefits of aggressive caching without the complexity of manual cache management or the consistency risks of simple TTL-based approaches. This might be especially beneficial for organisations at an intermediate level—not big enough to create bespoke caching solutions like the major tech firms have accomplished.

Conclusion

MuCache shows that automatic and consistent caching in microservices architectures is feasible without compromising performance or necessitating complicated application changes. Its thoughtful equilibrium of consistency assurances and performance enhancement, along with practical implementation factors, makes it an attractive solution to the difficult issue of caching in microservices.

As microservices architecture continues to evolve and become more complex, tools like MuCache can help to manage this complexity while maintaining performance, which will become increasingly important, whether through direct adoption of MuCache or incorporation of its ideas into other tools and frameworks.

References

[1] H. Zhang, K. Kallas, S. Pavlatos, and V. Liu, "A General Framework for Caching in Microservice Graphs," 2024, *USENIX Association*. Available: https://www.usenix.org/conference/nsdi24/presentation/zhang-haoran

[2] A. Tiwari, "A sidecar for your service mesh," 2017, *Abhishek Tiwari*. doi: 10.59350/89sdh-7xh23.

[3] X. Zhu *et al.*, "Dissecting Overheads of Service Mesh Sidecars," 2023, *Association for Computing Machinery*. doi: 10.1145/3620678.3624652.