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 prevalent in modern software systems, they've brought both tremendous benefits and significant challenges. One of the most pressing challenges has been maintaining performance at scale while dealing with complex service dependencies and network communication overhead. Today, I want to explore MuCache, an innovative 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 diving into MuCache, it's worth understanding why caching in microservices architecture is simultaneously critical and challenging. In a monolithic application, function calls between components are straightforward and fast. However, in a microservices architecture, these same interactions become network of calls, introducing significant latency and resource overhead. This 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 3 calls for many requests.

While caching is a powerful technique, it also presents several challenges in the context of microservices. Implementing caching in microservices architectures has traditionally been fraught with difficulties.

Cache Coherence

The most significant challenge is keeping caches coherent. Unlike traditional cache coherence where you're dealing with individual read/write operations, microservices depend on complex webs of downstream service calls and state changes. Every time data is updated in a service's database, all caches holding that data need to be updated or invalidated. This coordination is difficult, especially in a distributed microservice environment.

Consider the complexity of managing a social network's home timeline - a feature we're all familiar with from platforms like Twitter. The caching challenge here runs deeper than it might first appear. A user's timeline isn't just a simple list of posts; it's a carefully orchestrated composition of multiple data points and business rules. The cache invalidation puzzle becomes particularly fascinating when you consider all the different factors that could necessitate refreshing the cached timeline.

Think about all the moving parts: a followee adds a new post, another user updates their privacy settings, or a post that appeared in the timeline gets deleted. Even more intriguingly, the ranking algorithm might be tweaked to better promote certain types of content. What makes this especially challenging is that many of these changes originate from services that never directly interact with

the timeline generation service itself. They ripple through the system, affecting the timeline's validity through complex chains of dependencies.

But here's where it gets interesting: even when certain changes invalidate the final timeline response, portions of the underlying data might still be perfectly valid and cacheable. For instance, if a user changes their profile picture, we don't necessarily need to recompute their post content or invalidate cached follower relationships. This granularity in caching strategy can make the difference between a system that merely works and one that truly performs at scale. 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 faced with microservice caching challenges, developers today typically find themselves choosing between three suboptimal approaches.

First, they might roll up their sleeves and craft custom coherence protocols specific to their application. While this hands-on approach offers control, it often leads to brittle solutions that are riddled with edge cases and resist adaptation as systems evolve.

The second common path is to focus exclusively on caching at the backend storage layer. While this approach is straightforward, it misses a crucial opportunity: the ability to short-circuit lengthy call chains before they traverse the entire service graph. By the time a request reaches the backend, we've already incurred the overhead of multiple network hops and service invocations.

Finally, many teams simply 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 you lose the benefits of caching, too long and you risk 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 stayed away from inter-service caching. The complexity of maintaining cache coherence across a dynamic service graph has made this a particularly challenging problem to solve in a general way.

Key Requirements

When designing a caching solution for microservice architectures, we must carefully balance several critical requirements. First and foremost stands correctness - any caching system we implement must

act as a transparent performance optimization layer. It should never introduce new behaviors 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.

Equally crucial is the requirement for non-blocking operation with minimal overhead. In the high-stakes world of distributed systems, we can't afford to introduce delays on the critical path of request processing. A caching layer must operate with the lightness of a shadow, adding performance benefits without imposing significant costs. Any synchronisation or coordination should happen behind the scenes, away from the main 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 various factors - 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 tackles these challenges by providing a framework that automatically manages caching across microservice graphs while maintaining strong consistency guarantees. The key insight of MuCache is to push cache management off the critical path of request processing while still maintaining correctness through careful tracking of 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 own 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 only linearisable datastores are supported.

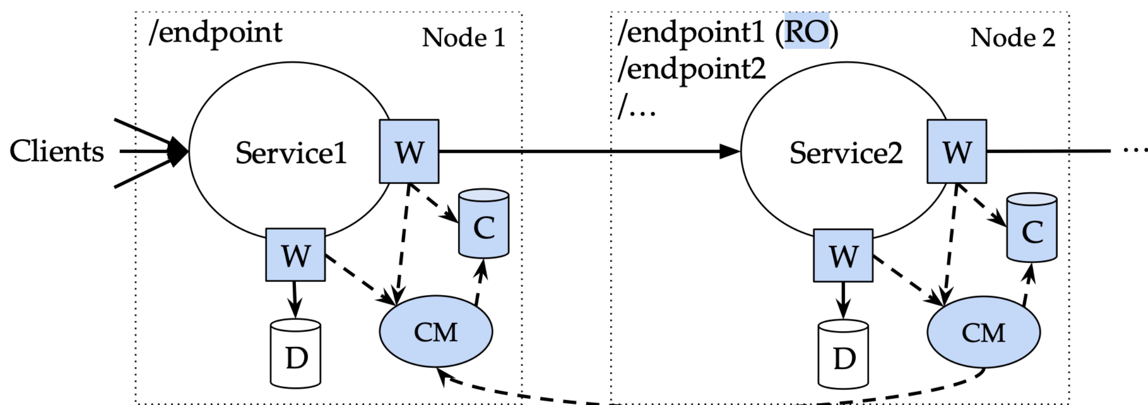

Figure 1: MuCache's Architecture. (C) denote caches, (CM) cache managers, (W) wrappers, and (D) the datastores. Wrappers are interceptor functions in the sidecar of each service. Solid arrows denote baseline communication while dashed arrows and blue com- ponents denote additions by our system. RO means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

What makes MuCache particularly interesting is its approach to consistency. Instead of trying to maintain strict consistency across all services (which would require expensive synchronization), it introduces a novel correctness condition based on client-observed behavior. This allows for reordering of operations between independent clients while still 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 cache their responses intelligently. Developers start by declaring which methods in their service APIs never modify 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, designed specifically for the dynamic nature of microservices graph. Mucache protocol brings two important features to the table. First, it eliminates the need for blocking operations during cache access. Whether you hit or miss the cache, your request keeps moving forward without waiting for other requests to complete. This design choice not only optimizes the happy path of cache hits but also ensures that cache misses incur minimal, predictable overhead.

The second is how MuCache maintains consistency. The protocol isn't just fast - it's provably correct, offering a powerful guarantee: any behavior you observe in a system with MuCache enabled could

have occurred in the original system without caching. This means 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 how it handles the complexity of 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's 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.

What's particularly clever about this design is how it handles 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 that has already been visited in the current request path. This prevents consistency violations that could occur from accessing stale data through different paths.

The MuCache prototype implementation comprises roughly 2k LoC of Go and authors have made it available on Github.

Performance and Real-World Impact

The performance improvements demonstrated by MuCache are impressive. In testing with real-world applications, it showed:

- 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's particularly noteworthy is how MuCache performs compared to both 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, but without sacrificing consistency guarantees.

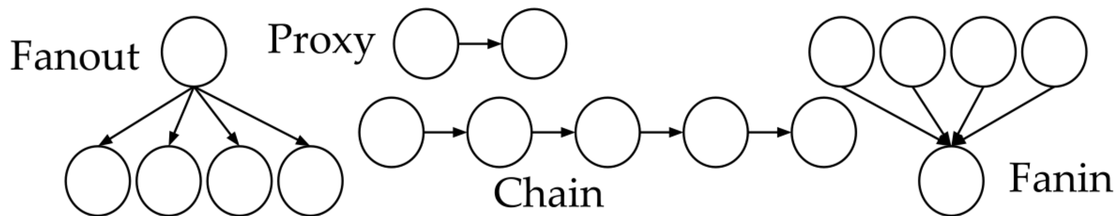

Figure 2: MuCache tested four synthetic patterns: proxy (simple two-tier), chain (sequential), fan-out (one-to-many), and fan-in (many-to-one). While proxy established baseline overhead metrics, chain patterns showed 2.6-3.1x lower median latency, fan-out improved tail latency by 1.6x, and fan-in achieved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

The system also shows excellent scalability characteristics. When testing with sharded services, MuCache maintained its performance advantages 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 clear of cross-shard synchronization.

Practical Considerations and Limitations

While MuCache represents a significant improvement in caching, it's important to understand its limitations and requirements.

Linearisable Datastores

The system requires linearisable datastores, which might not be available in all environments. It also needs to be able to intercept all service communications and database operations, which means it needs to be integrated with the service mesh layer.

Linearizability ensures that operations appear to take effect atomically at a single point in time between their invocation and completion. In practical terms, this means that once a write completes, all subsequent reads must see that write's effects.

While this requirement ensures strong consistency guarantees, it limits MuCache's compatibility with many popular datastores. For instance, many NoSQL databases like Cassandra or MongoDB, in their default configurations, offer eventual consistency rather than 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. This 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 2.5x reduction in median latency and 60% increase in throughput. This 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 consideration. 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 this overhead is minimal (less than 0.4% of cache size on average), systems with extremely high request rates or complex dependency graphs might need to tune the garbage collection parameters to maintain reasonable memory usage.

Cache Poisoning

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

In MuCache's context, the risk surface is amplified because cached results can be reused across different request paths. A compromised service could potentially inject malicious data into the cache, which would then be served to other services without re-validation. This could be particularly concerning because MuCache's design priorities performance by avoiding synchronous validation on the critical path.

Future Direction

MuCache represents a step forward in making microservices architectures more efficient and easier to manage. Its 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 careful tracking of dependencies while avoiding synchronization 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 could be particularly valuable for organizations in the middle ground - too large to stick with a monolithic architecture but not large enough to invest in building custom caching solutions like major tech companies have done.

Conclusion

MuCache demonstrates that it's possible to achieve automatic, consistent caching in microservices architectures without sacrificing performance or requiring complex application modifications. Its careful balance of consistency guarantees and performance optimisation, combined with practical implementation considerations, makes it a compelling approach for the challenging problem of microservices caching.

As microservices architecture continue to evolve and become more complex, tools like MuCache that help manage this complexity while maintaining performance will become increasingly important. Whether through direct adoption of MuCache or through incorporation of its ideas into other tools and framework.

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.