The Hidden Cost of Success: Understanding Non-Fatal Errors in Microservices

Abhishek Tiwari

Citation: *A. Tiwari*, ”The Hidden Cost of Success: Understanding Non-Fatal Errors in Microservices”, Abhishek Tiwari, 2024.
doi:10.59350/m7rzr-57k09

Published on: December 28, 2024

A recent study published by researchers from Washington University in St. Louis and Uber Technologies reveals a critical but often overlooked pattern in microservices architecture: non-fatal errors. While these errors do not cause system failures, they introduce noticeable performance overhead that can impact user experience and operational costs. The study by Lee et al. explores the prevalence of non-fatal errors and their impact on the observed latency of top-level requests.

What is a Non-Fatal Error?

At a technical level, in microservices architecture, a non-fatal error represents a unique scenario where internal operations fail without causing the overall client request to fail. For instance, a user requesting a personalised dashboard: even if the call to the personalisation service fails, the system can still return a default dashboard view to the user. From the user's perspective, the request succeeds with an HTTP 200 status code, but internally, the call chain encountered and handled failure gracefully. A non-fatal error is a fatal error not propagated back to the client by squashing or handling the error somewhere in the call chain. It is important to note that a non-fatal error is an intrinsic property of a service dependency graph or call chain (see [1]). Despite best efforts, there will always be some percentages of non-fatal errors, mainly when operating services with transactions per second (TPS) in millions.

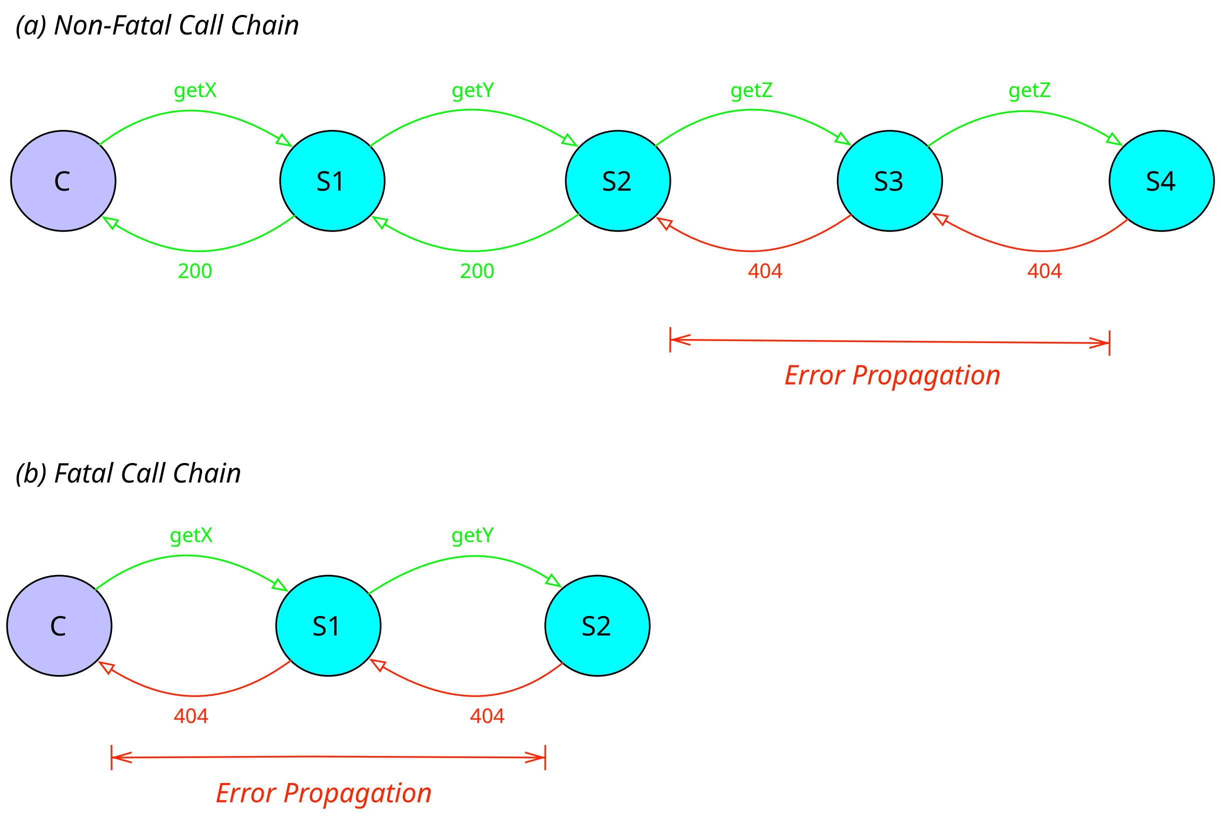

Figure 1: In a fatal call chain, client or caller (C) receives a 4xx error from Service S2, which Service S1 propagated. With a non-fatal call chain, although there was a fatal error somewhere deep in the call chain (in this case at Service S4, which Services S3 then propagated), one of the services (S2) in the call chain squashed the error, resulting in a 2xx response to the client.

This behaviour underscores a fundamental design principle of microservices: resilience through graceful failure handling. Services are engineered to handle failures gracefully, often through fallback mechanisms that enable them to continue functioning even when downstream services encounter issues. In some instances, services might employ static stability mechanisms such as local cache to manage the adverse response from their service dependencies. While this approach ensures the system's reliability, it can sometimes obscure underlying issues.

At a non-technical level, the definition of a non-fatal error varies by organisation. What works for Uber may not apply to CapitalOne.

Prevalence and Distribution

The research by Lee et al. revealed that non-fatal errors are prevalent in microservices architecture (see [2]). An analysis of 11 billion Remote Procedure Calls (RPCs) across 1,900 user-facing endpoints at Uber provides insight into how non-fatal errors affect large-scale microservices architecture. In their study, Lee et al. found that while only 0.65% of client requests fail (fatal error), approximately 29.35% of successful requests contained at least one non-fatal error in their call chain handled gracefully by the services.

Their data shows that 84% of endpoints encountered at least one non-fatal error during the study period. Some endpoints experienced non-fatal errors in over 95% of their requests, suggesting deeply embedded inefficiencies in certain services.

The Performance Impact

The actual cost of non-fatal errors becomes apparent when examining their impact on system performance and latency. The study found that requests containing non-fatal errors require 1.9 times more computational work than their error-free counterparts. This increased workload translated directly into increased latency observed by users, with non-fatal requests averaging 1.8 times higher latency than error-free requests.

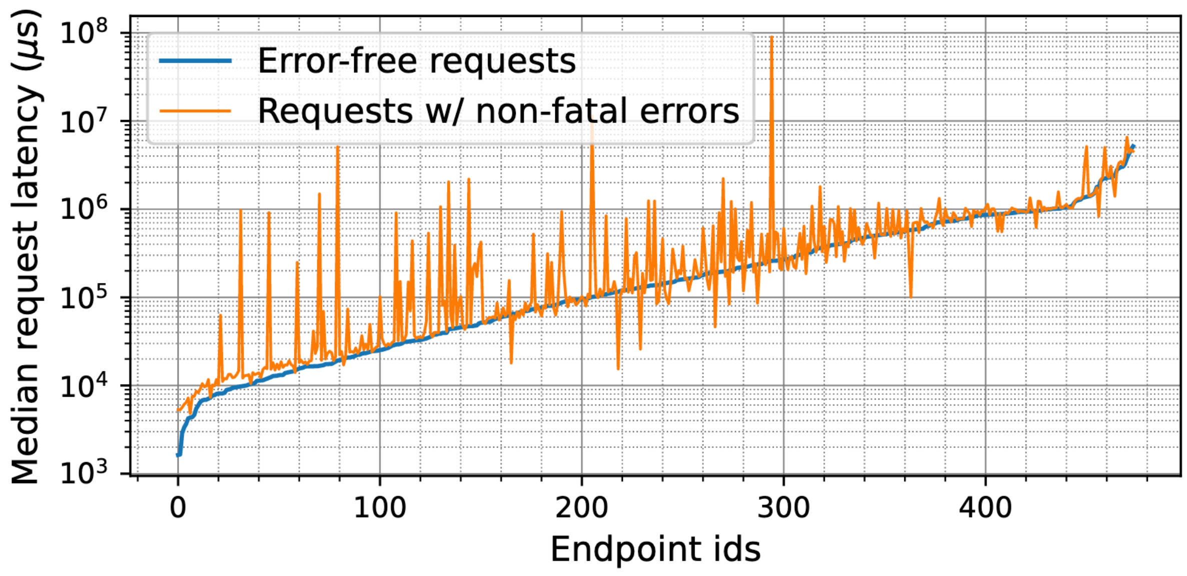

Figure 2: Latency is higher for requests with non-fatal errors compared to their error-free counterparts. Image credits Lee et al.

Non-fatal errors can cause a higher-than-usual increase in tail latency. The study found that the 99th percentile (P99) latency for requests with non-fatal errors was 2.9 times higher than for error-free requests. This degradation in P99 latency can directly impact user experience, particularly for time-sensitive operations.

Understanding Error Patterns

The research identified four primary categories of non-fatal errors in Uber's microservices ecosystem. All 4 of these can be generalised to any microservices ecosystem. **Entity Not Found errors** represented 42.46% of non-fatal errors, typically when services searched for data across distributed databases with inefficient lookup patterns. **Aborted Operations** made up 23.18% of errors, often related to concurrency issues and failed transactions. **Failed Preconditions** contributed 16.47%, usually caused by mismatched state assumptions or validations between services. **Resource Exhaustion** accounted for 16.29%, resulting from capacity limits and resource management issues.

Error propagation

The study suggests that non-fatal error propagation through microservices architecture follows a different pattern than fatal errors. Compared to fatal errors, non-fatal errors typically originate deeper in the RPC call chain. They had relatively short propagation paths, with over 80% contained within three or fewer levels of propagation.

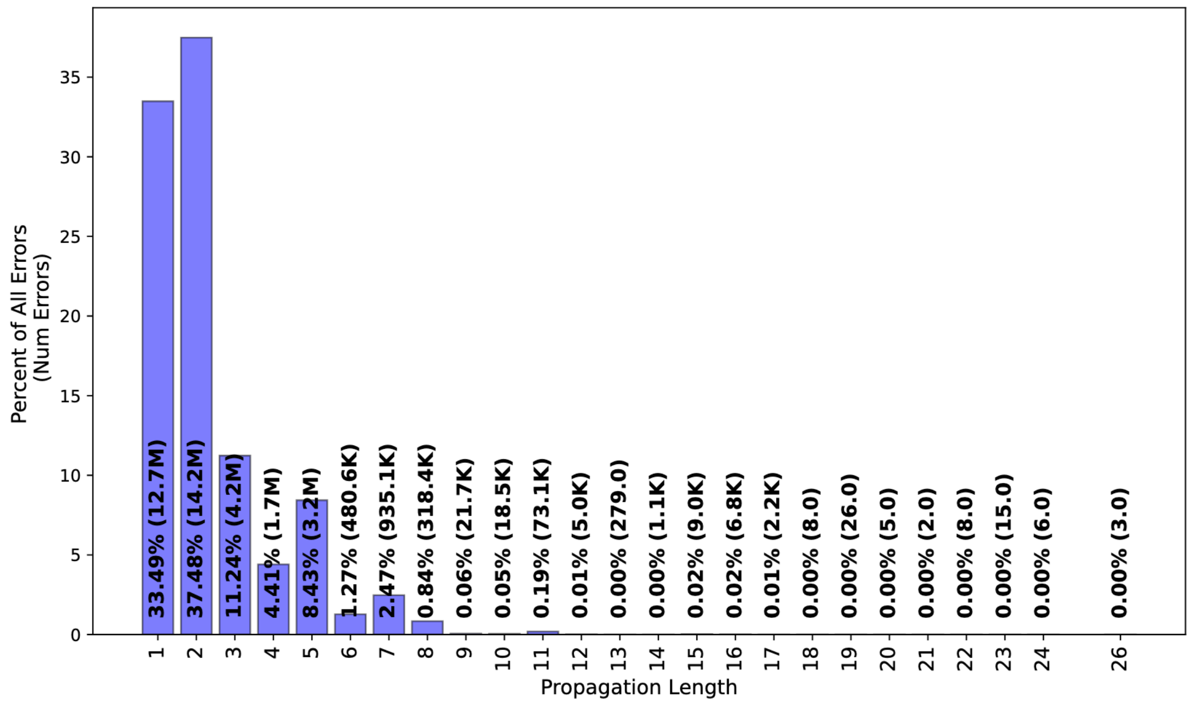

Figure 3: Propagation length of errors is defined as the difference between the depth of the RPC where the error originates and the depth of the caller RPC that squashes the said error. Image credits Lee et al.

Moreover, API error handling demonstrated a notable bimodal distribution: approximately 29% of APIs consistently stopped error propagation, while 63% always propagated errors. This suggests that most services implemented error-handling strategies, though they were not consistent.

Testing is hard

The prevalence of non-fatal errors reveals a key challenge in testing microservice architectures. In a microservices architecture with 6,000 services like Uber's, the number of possible call paths and error scenarios can grow exponentially. The complexity becomes evident if one considers how each service might call multiple downstream services, with each having different response patterns ranging from success to various errors. This complexity is further amplified by varying error-handling behaviours between services. As noted above, some opted to propagate errors while others handled them locally.

The challenge grows further with asynchronous call paths, the use of feature flags, and the constant evolution of services through different versions and deployments, each creating new permutations and combinations of possible call paths. Traditional end-to-end testing cannot cover all possible error scenarios for all possible call paths. Service dependencies make it difficult to reproduce specific error conditions, while timing-related issues often manifest only under actual production load.

This inherent complexity means that many non-fatal error scenarios are often discovered only in production, highlighting the need for robust monitoring and observability solutions. Organisations, particularly those with large-scale microservices ecosystems, should invest in production error analysis tools and develop consistent error-handling patterns. The focus should shift from attempting to test every possible call path to building services that can handle unexpected errors gracefully while maintaining visibility in these failures.

Identifying Non-Fatal Errors

The study discussed several methods to identify services and endpoints experiencing non-fatal errors.

Distributed Tracing Analysis

Distributed tracing frameworks such as Jaeger and OpenTelemetry can be used to track requests as they traverse through the microservices call graph. The study analysed distributed traces generated by Jaeger - Uber's distributed tracing framework - to find RPC errors that did not cause top-level failures but increased latency. Findings highlighted the following indicators as proxies for non-fatal errors:

- Endpoints where successful requests have higher latency than average
- Errors in deeper service call chain that do not propagate to the top
- Services showing high rates of fallback behaviours for data lookup

Critical Path Analysis

Analysing the critical path of requests helps to identify where non-fatal errors contribute most to latency in the call chain. The research found that not all errors impact latency equally - those on the critical path have the most significant effect. About 50% of errors occur off the critical path and, therefore, have minimal impact on latency. Distributed tracing is a prerequisite for the critical path analysis of microservices architecture.

Tail Latency Examination

The study suggests that endpoints showing noteworthy differences between median and tail (P99) latency indicate hidden non-fatal errors. The case of Uber requests with non-fatal errors showed 2.9x higher P99 latency compared to error-free requests, making tail latency analysis a valuable indicator of problematic endpoints.

These approaches, combined, provide a comprehensive way to identify endpoints with non-fatal errors.

Potential Optimisations

The study demonstrates that many non-fatal errors originate from suboptimal design patterns rather than unavoidable system conditions. By systematically addressing these patterns, organisations can improve performance and resource utilisation in their microservices architecture.

Eliminating Fallback Data Lookups

When a service encounters an “entity not found” error, it often triggers redundant or fallback lookups across multiple data stores. For example, in the user profile service described in the paper, when fetching user information, the system would check multiple regional databases sequentially (Americas, Europe, Asia-Pacific) even when user location data was available. By using available context to target the correct database directly, the service reduced latency by 30%.

Teams should examine error patterns to understand where similar redundant lookups occur and implement more intelligent routing strategies.

Removing Obsolete Feature Flags

The study reported several instances where Uber services continued to invoke deprecated features or used obsolete feature flags. For instance, the app-launch endpoint was still attempting to connect to a decommissioned pool-provider service, causing predictable failures that added unnecessary latency. Regular audits of dependencies and feature flags help surface and remove these defunct invocations, which consistently result in non-fatal errors.

Guidelines for Error Handling and Retry

To ensure consistent error handling across all services, organisations can define internal strategies for handling and propagating microservices errors. Teams should also document guidance on when a service can contain errors locally versus propagating them to upstream services or initiating the client.

Many services repeatedly attempt the same failed operation within a single request flow. The paper describes how the get-stores-view endpoint repeatedly called get-user despite previous failures in the same request chain. Implementing request-scoped error caching can ensure that once an operation fails, subsequent attempts within the same request can quickly return the previously cached error response rather than executing the whole operation again.

Finally, should retrying in request flow be completely discouraged to avoid cascading impact? Uncontrolled retries can cause retry storms in a large-scale microservices architecture. Approaches like exponential backoff, retry budgets, and circuit breakers can be explored to mitigate retry-related concerns.

Optimising Parallel Execution

The study found cases where Uber services made parallel calls to multiple backends when the data could have only existed in one location. For example, the fetchInfo API simultaneously queried mobile and web data stores even though data would only be present in one store.

While parallel execution can improve performance in some cases, it can also lead to unnecessary errors and resource consumption. Teams should make informed decisions about parallel vs. sequential execution not just based on performance considerations but also on factoring data locality and business requirements.

Reverse Context Propagation

Reverse Context Propagation involves passing error context and metadata back through the response chain, helping upstream services make more intelligent decisions about retries, error handling and propagation. Technically, one can use existing distributed tracing infrastructure to ensure all services in a call chain are aware of any fatal error and subsequent squashing or graceful handling of the error to make intelligent decisions.

Latency-Reduction Estimator

The study also introduces a new technique, the Latency-Reduction Estimator, to help predict latency improvements if non-fatal errors are eliminated. This method can analyse distributed traces to identify services and endpoints where non-fatal errors impact latency. The estimator can forecast performance gains by comparing traces with and without non-fatal errors, helping teams prioritise fixing non-fatal errors.

Business Definition

While non-fatal errors in contexts like ride-sharing or streaming services might translate into unacceptable or poor user experience, these errors in highly regulated industries like banking, healthcare, or insurance can have serious compliance and legal implications. For example, a KYC (Know Your Customer) verification step that falls back to a previous verification due to a non-fatal error could violate compliance requirements. A non-fatal error retrieving patient history that returns partial data could cause incomplete medical assessments. These examples highlight why error handling and monitoring in regulated industries must go beyond simple success/failure or system metrics. Organisations must carefully evaluate which errors can be treated as "non-fatal" and the actual cost of non-fatal errors and ensure their error handling and propagation strategies align with their regulatory obligation.

Conclusion

While non-fatal errors do not cause system failures, their influence on performance and user experience can not be ignored. Addressing non-fatal errors becomes critical to delivering a better user experience as the depth of the microservices call graph increases. Organisations with large-scale microservices ecosystems should consider implementing methods to identify and fix non-fatal errors.

Lastly, more research is required to understand the side-effects of non-fatal errors beyond performance, including overall debuggability and impact on an organisation's compliance posture.

References

[1] A. Tiwari, "Unveiling Graph Structures in Microservices: Service Dependency Graph, Call Graph, and Causal Graph," 2024, *Abhishek Tiwari*. doi: 10.59350/hkjz0-7fb09.

[2] I.-T. A. Lee, Z. Zhang, A. Parwal, and M. Chabbi, "The Tale of Errors in Microservices," 2024, *Association for Computing Machinery*. doi: 10.1145/3700436.