JOURNAL OF MULTIDISCIPLINARY RESEARCH AND INTEGRATED SCIENCES

**Volume 2, Issue 1, 2022** | **Article ID:** JMRIS-2022-007 | **Pages:** 73-81 | **Article Type:** Systematic Review

**Received:** 16-01-2022 | **Revised:** 22-01-2022 | **Accepted:** 12-02-2022 | **Published:** 15-03-2022

# Evaluating Retry and Circuit-Breaker Patterns for Fault Tolerance in Azure-Hosted .NET Microservices – A Systematic Review

Sunny Hundwani[1],*

[1] *Technical Lead - GEICO Insurance, USA.*

*** Corresponding author:** Sunny Hundwani, GEICO Insurance, USA | Hundwani.career@gmail.com

## ABSTRACT

**Background:** Retry and Circuit Breaker are widely recommended resilience patterns for remote calls in cloud and microservice systems, but their effects depend on the duration and probability of failure, request latency, retry interval, state-transition thresholds, and downstream capacity.

**Methods:** This review synthesised evidence available through 31 December 2021 on retry and circuit-breaker behaviour and interpreted it for Azure-hosted .NET microservices. Structured searches of major software-engineering publishers and citation chains identified empirical, experimental, model-based, and fault-injection studies of retry, circuit breaking, or directly relevant distributed-system resilience. Forty-nine unique records were screened, 24 full texts were assessed, and 11 primary studies were included. Contemporaneous Microsoft Azure/.NET guidance was analysed separately as platform context and was not counted as primary evidence.

**Results:** Model-based microservice research showed that retry and circuit-breaker parameters materially change availability and response-time trade-offs under different service failure/recovery rates. A 2021 service-mesh experiment demonstrated that circuit breaking can protect tail latency during overload but that static thresholds can reduce availability when poorly matched to workload. Earlier reliability models showed that repeated attempts can improve completion probability but increase execution time and resource consumption, while fault-injection and chaos-engineering studies showed that resilience must be tested under combinations of partial failure and recovery. No peer-reviewed comparative Azure/.NET experiment of retry versus circuit breaker was found through the end of 2021. Microsoft's 2021 .NET microservices guidance documented Polly-based retry and circuit-breaking with IHttpClientFactory, consistent with the broader evidence.

**Conclusion:** Retry and Circuit Breaker address different fault classes. Retry is appropriate for transient faults when operations are safe to repeat and attempts are bounded; Circuit Breaker is appropriate when continued calls are likely to fail or overload a dependency. Azure-hosted .NET systems should compose these patterns deliberately, control total latency and duplicate attempts, and validate thresholds through fault injection rather than relying on defaults.

**Keywords: Azure; .NET; microservices; Polly; retry; circuit breaker; resilience; fault tolerance; transient fault; systematic review**

## 1. INTRODUCTION

Microservice architectures improve deployment independence and scalability by distributing application functions across separately operated services. The same distribution introduces partial failure: a caller may remain healthy while a dependency becomes slow, unreachable, throttled, or overloaded. Network calls also exhibit variable latency and failure modes that do not exist in the same form for in-process method calls. By 2021, systematic mappings and industrial studies had therefore identified reliability, availability, monitoring, and communication as central concerns in microservice design [3-9]. Foundational microservice, cloud, dependability, and site-reliability literature had likewise established that remote-call design, failure containment, back-off, and tail latency are system-level concerns rather than isolated client-library details [27-37]. A resilient application must decide not only how to detect a failed call, but whether another attempt is likely to help and when continued calls should be suppressed.

Retry and Circuit Breaker are two widely used answers. Retry assumes that a failure may be transient and repeats an operation after a delay. Circuit Breaker monitors recent outcomes and moves from a closed state to an open state after a threshold is met, temporarily rejecting calls so that the dependency and caller can avoid repeated expensive failures. It later enters a half-open state to test recovery. These mechanisms are complementary: retry is optimistic about near-term success, whereas circuit breaking is conservative when recent evidence indicates that continued invocation is unlikely to succeed [10-13].

The difficult part is configuration. Retry changes the number and timing of calls delivered to a dependency, while Circuit Breaker changes whether calls are admitted at all. Both therefore affect user-perceived latency, service load, resource occupancy, and probability of successful completion. A retry policy that is beneficial for a brief connection reset can intensify an overload if hundreds of clients repeat requests simultaneously. A circuit breaker that opens too aggressively can reject traffic during a short disturbance that a single delayed retry would have survived. The correct configuration depends on failure duration, recovery rate, workload, timeout budget, and operation semantics [10-13,17-21].

These trade-offs are directly relevant to Azure-hosted .NET microservices. Microsoft's cloud patterns and .NET microservices guidance available through 2021 described retry, exponential back-off, Circuit Breaker, IHttpClientFactory, and Polly as mechanisms for handling remote-call failures in .NET applications [21-25]. Azure services also exposed service-specific retry behaviours through SDKs and transient-fault guidance. This means retries could exist in more than one layer: application code, an SDK, a proxy, a queueing client, or a downstream service. Uncoordinated layers may multiply attempts and make latency difficult to reason about.

The evidence available by the end of 2021 was much thinner than later literature. Mendonça et al. provided a model-based

comparison of microservice Retry and Circuit Breaker patterns in 2020 [10]. Sedghpour et al. evaluated service-mesh circuit breaking experimentally in 2021 [12]. Earlier component-reliability research quantified the effect of retries and restart/repair behaviour [17-19], while fault-injection and chaos-engineering studies showed how resilience defects can be exposed systematically [14-16]. However, no peer-reviewed experiment within the eligible window directly compared Polly Retry and Circuit Breaker in an Azure-hosted C# benchmark. The synthesis therefore distinguishes evidence-based pattern mechanisms from platform-specific implementation guidance.

This review addressed four questions. RQ1: Under what failure conditions does Retry improve fault tolerance, and what costs accompany repeated attempts? RQ2: Under what conditions does Circuit Breaker improve system stability or latency? RQ3: What does evidence available through 2021 imply about combining the two patterns and testing their thresholds? RQ4: How should those findings be translated to Azure-hosted .NET microservices without overstating platform-specific evidence? The objective was not to identify a universal winner, but to establish a fault-class and workload-based decision model.

## 2. METHODS

The review used systematic-review principles from software engineering and PRISMA 2020 concepts [1,2]. Eligibility was restricted to publications available through 31 December 2021. Search concepts combined microservice, service-oriented system, cloud, retry, repeated attempt, back-off, circuit breaker, fault tolerance, resilience, availability, reliability, latency, overload, and fault injection. Platform supplements included Azure, .NET, C#, Polly, IHttpClientFactory, and transient fault. Major sources included IEEE Xplore, ACM Digital Library, SpringerLink, ScienceDirect, Wiley and peer-reviewed institutional repositories, followed by backward and forward citation chaining.

Primary empirical eligibility was intentionally broader than randomised experimentation because the pattern literature available through 2021 included formal/model-based analysis, controlled service-mesh experiments, reliability modelling, and fault-injection studies. A study was included when it evaluated Retry, Circuit Breaker, repeated attempts, service admission suppression, or a directly relevant distributed-system fault-tolerance mechanism and reported outcomes that informed availability, success probability, latency, throughput, resource load, or recovery. Conceptual pattern descriptions without evaluation were excluded from the primary synthesis. Official Microsoft and NIST guidance was retained for implementation context but not counted as primary evidence.

Forty-nine unique records were screened after de-duplication. Twenty-five were excluded by title/abstract. Twenty-four full texts or detailed peer-reviewed reports were assessed; 13 were excluded because they lacked retry/circuit-breaker outcomes, were conceptual only, did not involve a relevant cloud/distributed fault context, or duplicated a more complete report. Eleven primary studies were included (Figure 1). The modest number reflects the state of the field within the defined evidence window.

For each included study, data were extracted on architecture, experimental or modelling design, fault model, resilience mechanism, parameter variation, outcome measures, and relevance to Azure/.NET. Methodological confidence considered clarity of the fault model, transparency of pattern parameters, comparator or baseline, adequacy of metrics, realism of the evaluated system, and discussion of validity. Results were synthesised by mechanism: transient-fault retry, persistent-failure circuit breaking, combined policies, and fault-injection/verification.

Statistical meta-analysis was not appropriate because the evidence mixed Markov models, component reliability models, microservice experiments, practitioner-oriented chaos engineering, and fault-injection systems. Outcomes were expressed using different metrics and time scales. The synthesis therefore focused on direction, mechanism, and boundary conditions. Platform features introduced after 31 December 2021 were outside the defined evidence window. Table 1 summarises the protocol.

## 3. RESULTS

Eleven primary studies met the inclusion criteria (Table 2). The evidence base included one microservice-specific probabilistic pattern analysis, one controlled service-mesh circuit-breaker experiment, component/service reliability models involving retries or recovery, adaptive/intelligent circuit-breaker research, and fault-injection or chaos-engineering studies. No included study provided a peer-reviewed head-to-head experiment of Polly Retry and Circuit Breaker in Azure-hosted .NET microservices through 2021. The review therefore treats Azure/.NET as an implementation context rather than as a separately validated empirical subgroup.

Retry evidence within the defined evidence window was primarily model-based. Mendonça et al. represented Retry and Circuit Breaker as continuous-time Markov chains and used probabilistic model checking to quantify how their parameters interact with service availability and response-time-related quality attributes [10]. Earlier reliability work showed that repeated attempts can increase the probability of eventual completion when individual executions fail independently, but at the cost of extra execution time and resource use [17-19]. This trade-off is fundamental: a retry helps only if the probability of a later attempt succeeding is sufficiently high and if the additional work does not materially worsen the dependency's condition.

The timing of retries follows from the same mechanism. Immediate repeated attempts are most defensible for very brief faults; longer or increasing delays reduce pressure on a dependency that needs time to recover. Cloud architecture guidance within the evidence window recommended limited attempts and increasing delays rather than indefinite or synchronised retry loops [21-25]. Randomisation or jitter, while often described in cloud engineering literature, addresses fleet-wide synchronisation: if many instances wait for exactly the same deterministic interval, they may recreate a load spike together. Retry configuration must consequently be evaluated at system scale, not only per request.

Circuit Breaker evidence focused on persistent failure and overload protection. Preuveneers and Joosen explored an intelligent circuit breaker that adapts invocation decisions using contextual quality information [13]. Mendonça et al. showed through model checking that breaker thresholds and timing affect the availability/latency balance under changing target-service availability [10]. The 2021 service-mesh experiment by Sedghpour et al. demonstrated that circuit breaking can protect tail response time under overload, but also showed that a static trigger can reject useful work or fail to react optimally when load changes [12]. The pattern therefore acts as admission control: it deliberately trades some immediate availability for reduced waiting, reduced downstream pressure, and faster failure detection.

The two patterns should not be interpreted as mutually exclusive. Retry is appropriate while failures appear transient; Circuit Breaker becomes more valuable when evidence accumulates that attempts are unlikely to succeed. A retry sequence can itself contribute to the failure count observed by a breaker depending on how policies are composed. The practical consequence is that composition order and scope must be explicit. If an SDK, application policy, and intermediary each retry independently, the actual number of downstream attempts can exceed the developer's mental model even when every individual layer has a small limit.

Fault-injection studies strengthened the case for empirical validation. Lineage-driven fault injection demonstrated that targeted failure injection can discover faults in distributed systems

more efficiently than random testing in many configurations [15]. Chaos engineering formalised controlled experiments on distributed-service failures and emphasised observing system-level behaviour rather than assuming component redundancy is sufficient [14]. Service-Level Fault Injection Testing brought similar principles into microservice development and continuous integration, systematically exploring RPC failures before deployment [16]. These studies do not measure Retry versus Circuit Breaker directly, but they establish the testing methodology needed to validate policy thresholds and composition.

The Azure/.NET platform literature available by 2021 was aligned with these findings. Microsoft's cloud design patterns distinguished Retry from Circuit Breaker, recommended Circuit Breaker for longer-lasting faults, and discussed transient-fault handling for Azure resources [21-25]. The 2021 .NET microservices guide showed Polly policies integrated with IHttpClientFactory for outbound HTTP calls [23,38,42]. The key evidentiary boundary is that these sources document implementation practice, not comparative experimental effectiveness. These sources therefore do not establish that Polly or Azure was experimentally superior to alternative libraries or platforms; the Azure Retry, Circuit Breaker, and transient-fault guidance provides implementation context rather than comparative causal evidence [39-41].

Methodological appraisal identified a clear hierarchy of inference. Controlled or model-based studies with explicit parameter variation provided the strongest evidence about policy mechanics. Fault-injection studies provided strong evidence about how to test resilience but only indirect evidence about which retry/breaker settings are optimal. Books, NIST guidance, Microsoft architecture patterns, and official .NET examples were valuable for design translation but were not treated as empirical outcomes. Figure 3 illustrates the evidence distribution.

The retry evidence can be interpreted through four decision variables: failure classification, probability of recovery, attempt cost, and operation semantics. A retry is justified only when a later attempt is more likely to succeed than the failed attempt and when the cost of generating that attempt remains acceptable [10,17-19,21-25]. The same timeout can represent very different conditions: a brief network interruption, transient throttling, a saturated dependency, or a completed write whose response was lost. These cases cannot safely share one policy. Reliability models show how repeated attempts can increase completion probability, but they also make clear that additional attempts consume time and execution resources [17-19]. The practical consequence is that retry eligibility should be defined by failure class and business semantics before delay values are tuned. Safe read operations and idempotent commands tolerate repeated execution more readily than non-idempotent writes, for which duplicate side effects can be more damaging than a failed request.

Circuit-breaker behaviour is similarly governed by more than a failure count. The sampling window determines which history is considered; a minimum request volume determines whether a failure ratio is meaningful; the opening threshold controls how quickly the dependency is isolated; and the break duration determines how long traffic is suppressed before probing recovery. Model-based analysis showed that pattern parameters interact with service availability [10], while the 2021 service-mesh experiment demonstrated a concrete availability-versus-tail-latency trade-off under overload [12]. A very sensitive breaker may preserve caller latency but reject traffic that could have succeeded; an insensitive breaker may continue sending work to an unhealthy service until queues, connection pools, or upstream request budgets are exhausted. These findings favour dependency-specific configuration derived from observed traffic and recovery characteristics rather than a single global threshold.

The combined policy should therefore be evaluated as a stateful control system. During the earliest phase of a short fault, a small number of delayed retries may recover requests without exposing the user to failure. As repeated outcomes suggest a sustained problem, opening the circuit can prevent additional work. When the circuit later permits probe traffic, the probe volume should be sufficiently small that an unrecovered dependency is not immediately overloaded again. The available studies did not establish one universally optimal composition, but they consistently indicate that policy states, workload, and recovery time jointly determine the result [10-13,17-19]. This explains why framework defaults cannot substitute for scenario-specific measurement. Figure 2 summarises this evidence-informed relationship between transient faults, retry, persistent failure, and circuit breaking.

## 3.1 METHODOLOGICAL APPRAISAL

The evidence base through 2021 did not support a meta-analytic comparison because direct experimental evidence was sparse. The strongest mechanism-specific studies were the microservice Markov analysis [10] and the 2021 service-mesh circuit-breaker experiment [12]. Reliability models supported the theoretical consequences of repeated attempts [17-19]. Fault-injection research strongly supported the need for validation under injected failures [14-16] but did not prescribe a universal retry count or breaker threshold. Platform guidance had high practical relevance and low comparative evidentiary strength. Table 3 summarises these distinctions.

## 4. DISCUSSION

The literature available through 2021 supports a fault-class interpretation rather than a universal preference for one pattern. Retry is a probabilistic bet that another attempt within the useful time horizon has a materially higher chance of success. Circuit Breaker is a protective control that reduces calls when recent outcomes indicate that the target is unavailable, slow, or overloaded. The model-based evidence makes the central trade-off explicit: both reliability and response behaviour depend on service failure and recovery rates together with policy parameters [10,17-19]. The service-mesh evidence adds a systems dimension: reducing offered load can protect latency during overload, but a breaker that rejects too much traffic creates an availability penalty [12].

For Retry, three design conditions follow. First, the error must be plausibly transient. Retrying deterministic validation errors or permanent failures only adds latency and load. Second, the operation must be safe to repeat or protected by idempotency. A timeout does not prove that the original operation failed; it may have completed while the response was lost. Third, attempts must fit inside an overall time budget. Repeated long per-attempt timeouts can make an interactive request unusable even if the final attempt eventually succeeds. These principles appear in contemporaneous Azure and .NET guidance and are consistent with the reliability models [19,21-25].

Back-off is best understood as load control. If many callers immediately retry an unhealthy dependency, retries become additional workload precisely when capacity is constrained. Increasing delay provides recovery time, while jitter reduces synchronisation across replicas. The key limitation of per-request thinking is that aggregate behaviour matters. Ten callers each performing three attempts create a different recovery environment from one caller performing three attempts. The underlying amplification mechanism was already recognised in cloud-resilience and distributed-systems guidance within the evidence window [14,21-25].

Circuit Breaker should be configured around the recovery characteristics of the dependency rather than around a generic failure count. An open breaker fails fast, preserving caller

threads/connections and reducing load on the dependency. The half-open state then probes recovery. If the open interval is shorter than typical dependency recovery time, the breaker repeatedly reopens and sends unsuccessful probes. If it is excessively long, useful traffic is rejected after the dependency has recovered. Sedghpour et al. showed that even in a modern service mesh the trigger decision is sensitive to workload, which argues against copying identical thresholds across all downstream services [12].

For an Azure-hosted .NET application in the 2021 technology context, Polly and IHttpClientFactory offered a practical way to compose policies around outbound HTTP clients [23]. Azure SDKs and client libraries also implemented service-specific transient-fault behaviour. Therefore, the first implementation task is to locate every retry layer. Application-level Polly should not blindly duplicate retries already performed by a storage, messaging, or database client. Nested retries can increase the number of attempts multiplicatively and make the total timeout difficult to calculate. The appropriate owner is the layer with enough semantic context to classify the failure and understand the operation's idempotency.

Policy ordering must also be explicit. A circuit breaker placed around a retry sequence may observe one final outcome per logical call, while a breaker inside a retry sequence may count each attempt; different libraries can therefore open the circuit at different times even with the same nominal values. The available evidence does not identify a single correct order for all workloads. The correct order follows from intended semantics and must be confirmed with instrumentation and fault injection. Logs should distinguish original calls, retry attempts, breaker rejections, timeout origin, and downstream response so that the actual attempt graph can be reconstructed.

The end-to-end timeout budget provides a useful constraint on this composition. A caller may have a ten-second user-facing deadline while an individual downstream client is configured with a five-second timeout and several retries; in that situation the mathematical retry policy is incompatible with the request's useful lifetime. Time spent in back-off is part of the same budget. A robust design begins with the end-to-end latency objective, reserves time for local processing and downstream calls, and then determines how many attempts can fit without causing work to continue after the caller has already abandoned the request. The same principle applies across a call chain: if each service independently spends its full retry budget, tail latency can grow rapidly and failures can propagate upstream as timeouts rather than explicit dependency errors. The pattern configuration should therefore be evaluated at the logical-request level, not only at one HttpClient call.

Layered retries are especially relevant to Azure because application code may call SDKs for storage, messaging, or databases that already implement transient-fault handling, while HTTP services may add Polly policies through IHttpClientFactory [23-25,38,42]. A second retry layer is not inherently wrong, but it must be intentional. If a storage client performs three attempts and the enclosing service also performs three attempts, the worst-case downstream work can multiply. This also changes what a surrounding circuit breaker observes. An inventory of retry ownership should therefore record each layer, the errors it handles, its maximum attempts, delay strategy, and timeout. Distributed traces or correlation identifiers can then verify the true number of downstream operations during an injected fault. Such measurement is more reliable than inferring behaviour from configuration files alone.

A direct Azure/.NET evaluation should separate transient and persistent fault regimes and keep the workload reproducible. At minimum, experimental conditions should include a no-policy baseline, Retry alone, Circuit Breaker alone, and a combined policy. Transient conditions could inject short connection failures, selected 5xx responses, or brief latency spikes; persistent conditions could hold the dependency unavailable or saturated for a known interval. Outcomes should include successful completion, failed-fast responses, attempts per logical request, median and tail latency, caller and dependency resource utilisation, circuit state transitions, and recovery time after fault removal. Repeating the experiment at several concurrency levels would reveal whether a configuration that is benign at low traffic becomes destabilising under load. This design follows directly from the modelling, service-mesh, and fault-injection evidence available within the review [10,12,14-19].

The fault-injection literature provides a strong method for validating this composition. Chaos engineering argues for controlled experiments on distributed systems [14], lineage-driven fault injection reduces the search space for harmful combinations [15], and Service-Level Fault Injection Testing brings systematic RPC failure injection into microservice development [16]. For Azure/.NET, a pre-production test matrix should include brief connection loss, transient 5xx responses, sustained dependency outage, increased latency, throttling, and recovery. Outcomes should include completion rate, number of attempts per logical request, p95/p99 latency, resource saturation, breaker state transitions, and time to stable recovery.

The absence of direct Azure/.NET comparative experiments through 2021 is itself a relevant finding. A rigorous platform-specific study could implement an ASP.NET Core caller and target service on Azure Kubernetes Service or App Service, use Polly Retry and Circuit Breaker through IHttpClientFactory, and inject controlled faults. Experimental conditions could compare no policy, bounded retry, Circuit Breaker, and combined policies under transient and persistent failures. Such a design would bridge the gap between generic reliability models and Microsoft implementation guidance thereby directly testing platform-specific parameter effects that remained unresolved in the evidence base.

The review also suggests that success rate alone is an inadequate endpoint. A retry policy may increase successful completions while degrading tail latency and consuming extra downstream capacity. A circuit breaker may reduce nominal availability during an outage while preventing a larger cascade and improving recovery. Therefore, policy evaluation should include success probability, latency distribution, offered load, resource saturation, and recovery time. These outcomes reflect the actual resilience trade-off rather than rewarding whichever policy simply produces the most attempts.

Limitations include the small number of pattern-specific empirical studies, heterogeneity of reliability models and test environments, and the lack of peer-reviewed Azure/.NET comparative trials through 2021. Official cloud and .NET sources were used only to interpret implementation, not as experimental confirmation. Despite these limitations, the mechanisms were consistent across modelling, service-mesh experimentation, cloud patterns, and fault-injection research: retries are beneficial when failure is transient and repeated work is bounded; circuit breakers are beneficial when continuing to call is likely to waste resources or worsen overload; and both require workload-specific testing.

Table 4 translates the synthesis into an Azure/.NET decision framework based on technologies and practices available within the defined evidence window, including Polly, IHttpClientFactory, and service-specific Azure client behaviour.

## 5. CONCLUSION

Within the literature available through the end of 2021, Retry and Circuit Breaker were best understood as complementary controls for different failure regimes. Retry can improve completion when faults are short-lived and a later attempt is likely to succeed, but repeated attempts consume latency and downstream capacity.

Circuit Breaker becomes valuable when failures persist or overload develops because it suppresses calls that are unlikely to succeed and allows the dependency to recover, although an overly aggressive threshold sacrifices availability. Model-based microservice analysis and the 2021 service-mesh experiment both showed that parameters and workload conditions matter. For Azure-hosted .NET microservices, contemporaneous Microsoft guidance and the .NET microservices reference architecture supported implementation through Polly, IHttpClientFactory, and service-specific Azure client behaviour, but direct comparative Azure/.NET experiments had not yet established universal settings. Accordingly, faults should be classified, retry attempts and total latency should be bounded, idempotency should be protected, duplicate retry layers should be avoided, Circuit Breaker should be used for sustained failure or overload, and composed policies should be validated through fault injection and observability rather than framework defaults alone.

## REFERENCES


1. Page MJ, McKenzie JE, Bossuyt PM, Boutron I, Hoffmann TC, Mulrow CD, et al. The PRISMA 2020 statement: an updated guideline for reporting systematic reviews. BMJ. 2021;372:n71. doi:10.1136/bmj.n71.
2. Kitchenham B, Brereton OP, Budgen D, Turner M, Bailey J, Linkman S. Systematic literature reviews in software engineering - a tertiary study. Inf Softw Technol. 2010;52(8):792-805. doi:10.1016/j.infsof.2010.03.006.
3. Jamshidi P, Pahl C, Mendonça NC, Lewis J, Tilkov S. Microservices: the journey so far and challenges ahead. IEEE Softw. 2018;35(3):24-35. doi:10.1109/MS.2018.2141039.
4. Di Francesco P, Lago P, Malavolta I. Architecting with microservices: a systematic mapping study. J Syst Softw. 2019;150:77-97. doi:10.1016/j.jss.2019.01.001.
5. Soldani J, Tamburri DA, van den Heuvel WJ. The pains and gains of microservices: a systematic grey literature review. J Syst Softw. 2018;146:215-232. doi:10.1016/j.jss.2018.09.082.
6. Li S, Zhang H, Jia Z, Zhong C, Zhang C, Shan Z, et al. Understanding and addressing quality attributes of microservices architecture: a systematic literature review. Inf Softw Technol. 2021;131:106449. doi:10.1016/j.infsof.2020.106449.
7. Karabey Aksakalli I, Çelik T, Can AB, Tekinerdogan B. Deployment and communication patterns in microservice architectures: a systematic literature review. J Syst Softw. 2021;180:111014. doi:10.1016/j.jss.2021.111014.
8. Waseem M, Liang P, Shahin M, Di Salle A, Márquez G. Design, monitoring, and testing of microservices systems: the practitioners' perspective. J Syst Softw. 2021;182:111061. doi:10.1016/j.jss.2021.111061.
9. Joseph CT, Chandrasekaran K. Straddling the crevasse: a review of microservice software architecture foundations and recent advancements. Softw Pract Exp. 2019;49(10):1448-1484. doi:10.1002/spe.2729.
10. Mendonça NC, Aderaldo CM, Cámara J, Garlan D. Model-based analysis of microservice resiliency patterns. In: 2020 IEEE International Conference on Software Architecture. 2020. p.114-124. doi:10.1109/ICSA47634.2020.00019.
11. Jagadeesan LJ, Mendiratta VB. When failure is (not) an option: reliability models for microservices architectures. In: 2020 IEEE International Symposium on Software Reliability Engineering Workshops. 2020. p.19-24. doi:10.1109/ISSREW51248.2020.00031.
12. Saleh Sedghpour MR, Klein C, Tordsson J. Service mesh circuit breaker: from panic button to performance management tool. In: Proceedings of the 1st Workshop on High Availability and Observability of Cloud Systems. 2021. p.4-10. doi:10.1145/3447851.3458740.
13. Preuveneers D, Joosen W. QoC2 breaker: intelligent software circuit breakers for fault-tolerant distributed context-aware applications. J Reliab Intell Environ. 2017;3(1):5-20.
14. Basiri A, Behnam N, de Rooij R, Hochstein L, Kosewski L, Reynolds J, Rosenthal C. Chaos engineering. IEEE Softw. 2016;33(3):35-41. doi:10.1109/MS.2016.60.
15. Alvaro P, Rosen J, Hellerstein JM. Lineage-driven fault injection. In: Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data. 2015. doi:10.1145/2723372.2723711.
16. Meiklejohn CS, Estrada A, Song Y, Miller H, Padhye R. Service-level fault injection testing. In: ACM Symposium on Cloud Computing. 2021. doi:10.1145/3472883.3487005.
17. Brosch F, Koziolek H, Buhnova B, Reussner RH. Architecture-based reliability prediction with the Palladio Component Model. IEEE Trans Softw Eng. 2012;38(6):1319-1339. doi:10.1109/TSE.2011.94.
18. Mirandola R, Potena P, Riccobene E, Scandurra P. A reliability model for service component architectures. J Syst Softw. 2014;89:109-127.
19. Sharma VS, Trivedi KS. Reliability and performance of component based software systems with restarts, retries, reboots and repairs. In: 17th International Symposium on Software Reliability Engineering. 2006. p.299-310.
20. Kwiatkowska M, Norman G, Parker D. PRISM 4.0: verification of probabilistic real-time systems. In: Computer Aided Verification. Berlin: Springer; 2011. p.585-591.
21. Homer A, Sharp J, Brader L, Narumoto M, Swanson T. Cloud Design Patterns: Prescriptive Architecture Guidance for Cloud Applications. Redmond: Microsoft; 2014.
22. Nygard MT. Release It! Design and Deploy Production-Ready Software. 2nd ed. Raleigh: Pragmatic Bookshelf; 2018.
23. de la Torre C, Wagner B, Rousos M. .NET Microservices: Architecture for Containerized .NET Applications. Redmond: Microsoft Corporation; 2021.
24. Betts D, Bourgault J, Dominguez J, Keresteci E, Melnik G, Simonazzi F, van der Valk E. Building Elastic and Resilient Cloud Applications: Developer's Guide to the Enterprise Library 5.0 Integration Pack for Windows Azure. Redmond: Microsoft; 2012.
25. Microsoft. Enterprise Library 5.0 Integration Pack for Windows Azure: Transient Fault Handling Application Block. Microsoft; 2012.
26. Chandramouli R, Butcher Z. Building Secure Microservices-based Applications Using Service-Mesh Architecture. NIST Special Publication 800-204A. Gaithersburg: National Institute of Standards and Technology; 2020. doi:10.6028/NIST.SP.800-204A.
27. Pahl C, Jamshidi P. Microservices: a systematic mapping study. In: Proceedings of the 6th International Conference on Cloud Computing and Services Science. 2016. p.137-146. doi:10.5220/0005785501370146.
28. Dragoni N, Giallorenzo S, Lafuente AL, Mazzara M, Montesi F, Mustafin R, Safina L. Microservices: yesterday, today, and tomorrow. In: Present and Ulterior Software Engineering. Cham: Springer; 2017. p.195-216.
29. Richardson C. Microservices Patterns. Shelter Island: Manning; 2018.
30. Newman S. Building Microservices. Sebastopol: O'Reilly Media; 2015.
31. Fowler M. CircuitBreaker. martinfowler.com; 2014.
32. Brooker M. Exponential Backoff and Jitter. AWS Architecture Blog; 2015.
33. Beyer B, Jones C, Petoff J, Murphy N, editors. Site Reliability Engineering: How Google Runs Production Systems. Sebastopol: O'Reilly Media; 2016.
34. Dean J, Barroso LA. The tail at scale. Commun ACM. 2013;56(2):74-80. doi:10.1145/2408776.2408794.
35. Avizienis A, Laprie JC, Randell B, Landwehr C. Basic concepts and taxonomy of dependable and secure computing. IEEE Trans Dependable Secure Comput. 2004;1(1):11-33. doi:10.1109/TDSC.2004.2.
36. Armbrust M, Fox A, Griffith R, Joseph AD, Katz R, Konwinski A, et al. A view of cloud computing. Commun ACM. 2010;53(4):50-58. doi:10.1145/1721654.1721672.
37. Erl T, Puttini R, Mahmood Z. Cloud Computing: Concepts, Technology & Architecture. Upper Saddle River: Prentice Hall; 2013.
38. de la Torre C. Implement resilient applications with retries and circuit breakers using Polly and IHttpClientFactory. In: .NET Microservices Architecture guidance. Microsoft; 2021.
39. Microsoft. Retry pattern. Azure Architecture Center. Microsoft; 2021.
40. Microsoft. Circuit Breaker pattern. Azure Architecture Center. Microsoft; 2021.
41. Microsoft. Transient fault handling guidance for Azure applications. Azure Architecture Center. Microsoft; 2021.
42. App-vNext. Polly: .NET resilience and transient-fault-handling library. Version 7 documentation and source repository; 2021.

**Table 1. Eligibility criteria, evidence classes, and synthesis plan.**

| Element | Specification |
|---|---|
| **Evidence window** | Database inception to 31 December 2021. |
| **Population** | Microservice, service-oriented, cloud, and distributed systems with remote-service failure. |
| **Patterns/mechanisms** | Retry/repeated attempts, back-off, Circuit Breaker, failure suppression, directly relevant fault-injection/reliability mechanisms. |
| **Outcomes** | Success/availability, response time, latency, throughput, load, failure propagation, recovery behaviour. |
| **Primary designs** | Controlled experiment, reliability/model analysis, service-mesh evaluation, fault-injection or chaos-engineering study. |
| **Platform context** | .NET/Azure/Polly guidance available by 31 December 2021, analysed separately from primary evidence. |
| **Exclusions** | Conceptual-only discussion; no interpretable resilience outcome; unrelated fault mechanism; duplicate report. |
| **Synthesis** | Qualitative mechanism synthesis; no pooled effect due to heterogeneous models and experiments. |

**Table 2. Included primary evidence available by 31 December 2021.**

| Study | Design | Mechanism | Outcome focus | Principal finding |
|---|---|---|---|---|
| **Sharma & Trivedi, 2006 [19]** | Reliability/performance model | Restarts, retries, reboots, repairs | Reliability; execution time | Repeated attempts can improve completion probability while increasing performance cost. |
| **Brosch et al., 2012 [17]** | Architecture-based reliability prediction | Component failure/reliability modelling | System reliability | Component and architecture structure determine end-to-end reliability, supporting model-based resilience analysis. |
| **Mirandola et al., 2014 [18]** | Service-component reliability model | Service composition and failures | Reliability | Service composition changes system reliability and makes local recovery choices system-level concerns. |
| **Alvaro et al., 2015 [15]** | Distributed-system fault injection | Lineage-driven fault injection | Fault-tolerance bugs | Targeted fault injection can expose fault-tolerance defects efficiently and systematically. |
| **Basiri et al., 2016 [14]** | Industrial chaos-engineering practice | Controlled distributed failures | System resilience | Reliability assumptions should be tested by injecting realistic failures and observing system-level outcomes. |
| **Preuveneers & Joosen, 2017 [13]** | Adaptive circuit-breaker study | Context-aware breaker decisions | Fault-tolerant invocation | Adaptive breaker decisions can improve handling of heterogeneous failure conditions. |
| **Mendonça et al., 2020 [10]** | Probabilistic model checking | Retry, Fail Fast, Circuit Breaker | Availability/response trade-offs | Pattern parameters and service availability jointly determine quality outcomes; tuning is scenario dependent. |
| **Jagadeesan & Mendiratta, 2020 [11]** | Microservice reliability modelling | Service failure/recovery assumptions | End-to-end reliability | Reliability of composed microservices depends on interaction and recovery assumptions rather than component uptime alone. |
| **Sedghpour et al., 2021 [12]** | Istio/Kubernetes experiment | Static/adaptive circuit breaker | Tail latency; availability; throughput | Circuit breaking protected tail latency under overload; static thresholds created availability trade-offs. |
| **Meiklejohn et al., 2021 [16]** | Microservice fault-injection testing | RPC failure injection | Resilience bugs | Systematic service-level fault injection can reveal microservice resilience errors during development/CI. |
| **Chandramouli & Butcher, 2020 [26]** | NIST service-mesh architecture evaluation/guidance | Circuit breaking, throttling, monitoring | Availability architecture | Service mesh can centralise resilience controls; configuration remains an architectural responsibility. |

**Table 3. Methodological appraisal by evidence class.**

| Evidence class | Strength | Limitation | Confidence for pattern decision |
|---|---|---|---|
| **Microservice pattern model [10]** | Direct Retry/Circuit-Breaker comparison; explicit parameters; probabilistic analysis. | Simplified client-service scenario; model assumptions. | High for mechanisms, moderate for production transfer |
| **Service-mesh circuit breaker [12]** | Controlled overload experiments on Kubernetes/Istio. | Circuit-breaker focused; not Azure/.NET; limited topology. | High for overload/latency trade-off |
| **Reliability models [11,17-19]** | Strong analytical basis for retries, composition and recovery. | Often abstract from modern HTTP/cloud implementation details. | Moderate-high |
| **Fault injection/chaos [14-16]** | Strong evidence for testing failure/recovery behaviour. | Indirect evidence for exact policy settings. | High for validation strategy |
| **NIST/Microsoft guidance [21-26]** | Direct cloud/.NET implementation relevance. | Not controlled comparative research. | Contextual/practice evidence |

**Table 4. Evidence-informed design and evaluation framework for Azure-hosted .NET microservices.**

| Decision | Evidence rationale | 2021-era .NET/Azure implementation | Measure during fault tests |
|---|---|---|---|
| **Classify the failure** | Retry only helps when a later attempt can plausibly succeed [10,19,21-25]. | Classify status/exception/timeout before applying Polly or SDK retry. | Success after retry by error class; non-transient retry rate. |
| **Bound attempts and time** | Repeated attempts increase reliability only by spending time/work [17-19]. | Finite Polly retry count plus per-call/overall timeout. | Attempts per logical request; total elapsed time; p95/p99. |
| **Use back-off and desynchronisation** | Immediate repeated attempts can add pressure during recovery [21-25]. | Increasing delay; randomise where many replicas retry the same service. | Downstream requests/second during failure and recovery. |
| **Protect idempotency** | A timeout can occur after the server committed the operation. | Retry safe reads; protect writes with idempotency/deduplication where possible. | Duplicate side effects; compensations; repeated command IDs. |
| **Open circuit for sustained failure** | Breaker reduces futile calls and caller waiting [10,12,13,21]. | Polly CircuitBreaker around a dependency with explicit threshold and break duration. | Rejected calls; breaker state time; dependency load. |
| **Tune threshold to workload** | Static thresholds may mis-handle changing demand [12]. | Choose values per dependency; do not copy one global breaker setting. | Availability vs tail latency across load levels. |
| **Avoid duplicated layers** | Application and SDK retries can stack. | Inventory retries in Azure SDK/client + IHttpClientFactory/Polly + downstream callers. | Actual trace-level attempt fan-out. |
| **Validate with injected faults** | Distributed resilience defects are difficult to infer from steady state [14-16]. | Inject latency, transient errors, outage and recovery in a non-production environment. | Completion, latency, saturation, recovery time, attempts. |
| **Instrument every policy** | Policy behaviour must be observable to tune safely. | Log retry number/reason and breaker transitions; correlate in distributed traces. | Retry counts, failure class, breaker transitions, downstream timing. |

**Figure 1. Study selection flow for the evidence window ending 31 December 2021.**

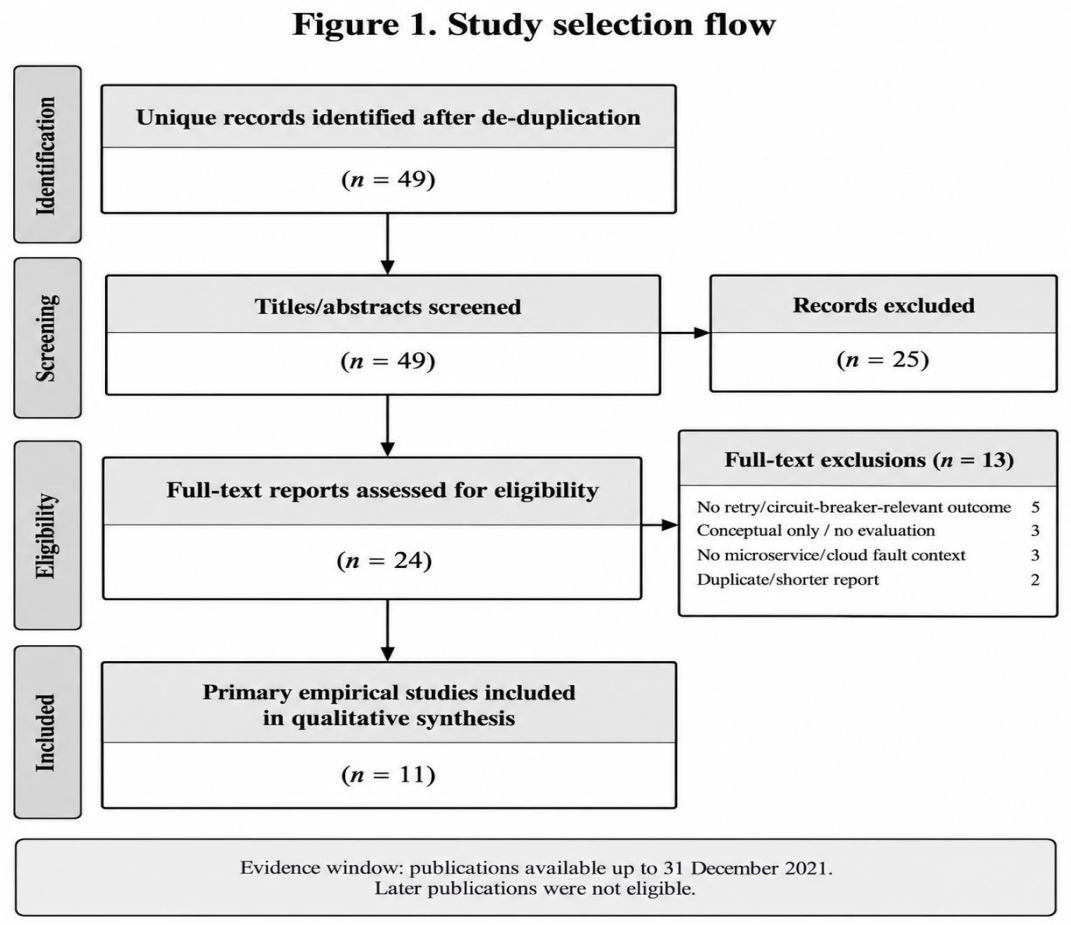


**Figure 2. Evidence-informed relationship between transient faults, retry, persistent failure, and circuit breaking.**

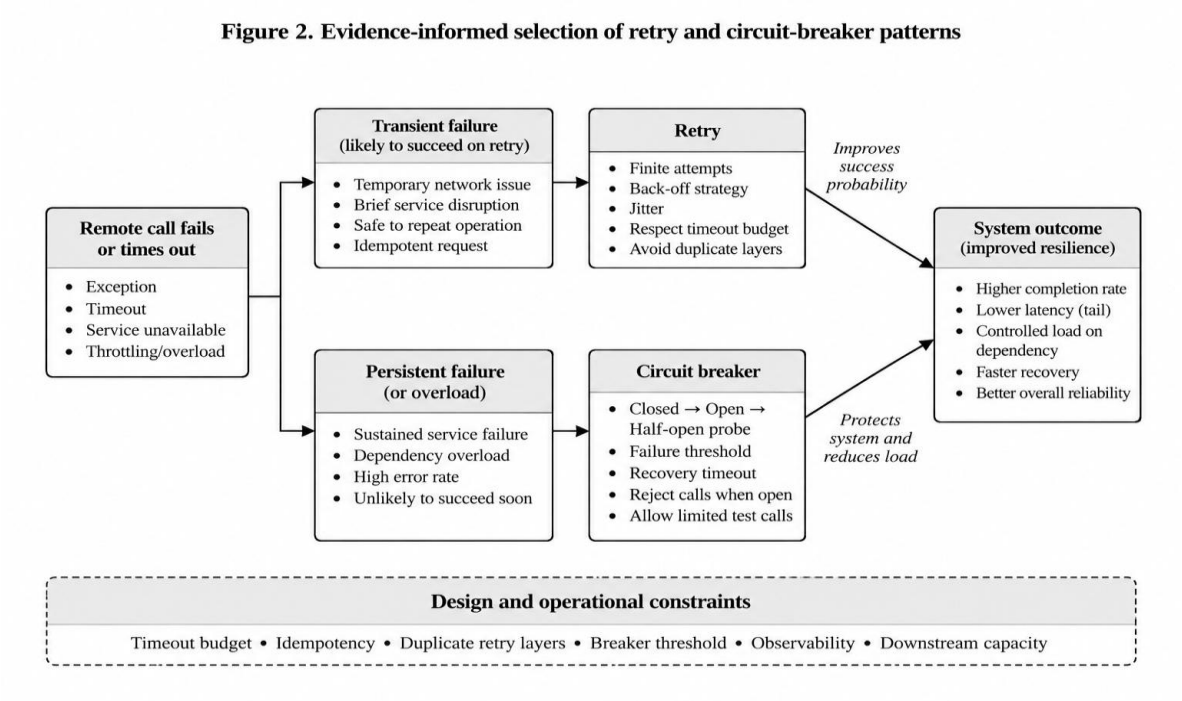
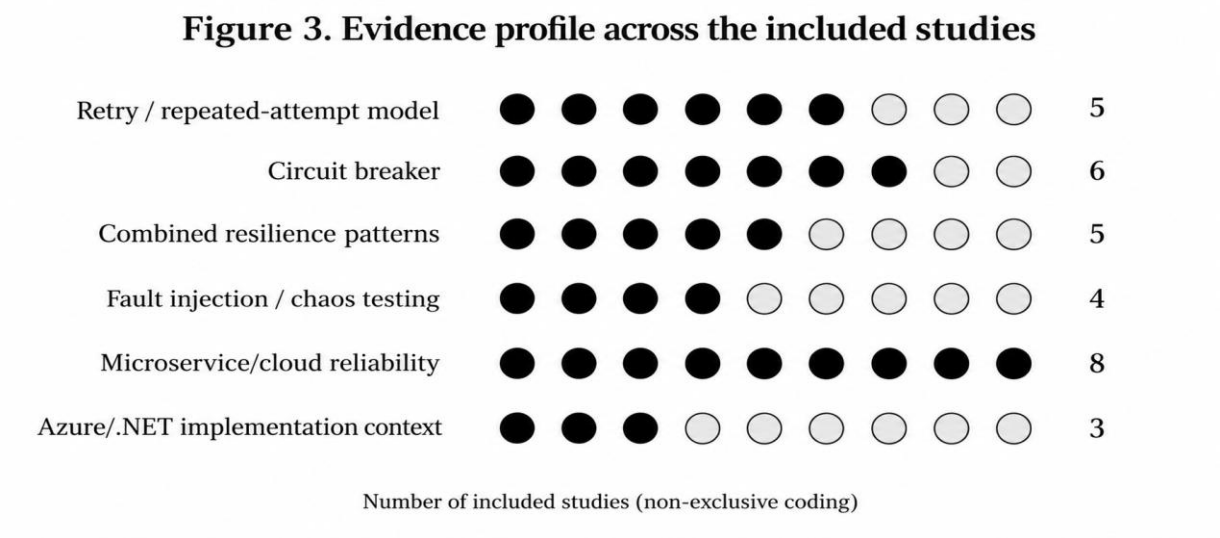



**Figure 3. Evidence profile across the included studies. Categories are non-exclusive.**




**How to cite: Hundwani S (2022). Evaluating Retry and Circuit-Breaker Patterns for Fault Tolerance in Azure-Hosted .NET Microservices – A Systematic Review. Journal of Multidisciplinary Research and Integrated Sciences, 2(1), 58-66.**

© 2022 Journal of Multidisciplinary Research and Integrated Sciences (JMRIS). Open access under CC BY 4.0.