International Journal of Computer Science and Engineering Development
Volume 1, Issue 5 | November - December 2026 | www.ijcsed.com
ISSN: **3139-0862**

Quantitative Reliability Assessment of Distributed Cloud Services: Common-Cause Failure, Retry Amplification, Error Budgets and Recovery-Time Decomposition

[1*]Raghvendra Rao, [2*] Dr. Yashpal Joshi
[1*] Phd. Research Scholar, Department of Computer Science and Engineering, Saveetha Amarawati University, Vijayawada, Andhra Pradesh, India
[2*] PhD Guide, Department of Computer Science and Engineering, Saveetha Amarawati University, Vijayawada, Andhra Pradesh, India

Abstract:
Reliability claims for distributed cloud services are often expressed as single availability figures that hide the assumptions behind them. This paper presents a transparent assessment method that models a service from the user journey, states every numerical input as an assumption and shows how the result changes when that assumption changes. Five analyses are developed from worked examples in a doctoral thesis. Three mandatory dependencies at 99.9% each give a journey availability of 99.70%, and twenty give 98.02%. Two independent paths that each fail 0.1% of the time give 99.9999% availability, but a common cause present 0.05% of the time lowers this to 99.9499%, raising expected unavailability from 0.04 to 21.6 minutes per 30 days. Three layers that each retry three times can multiply load by up to 27 in the worst case and by 5.4 on average when half of calls fail. In a sequential recovery of 18 minutes, halving restoration saves 4 minutes, while halving all other stages saves 5. At 100 acknowledged writes per second, a 20-second unrecoverable interval exposes 2,000 writes. The results show that independence assumptions, retry policies and non-technical recovery stages dominate reliability outcomes, and that each analysis must be validated with measured data before use. All values are calculated illustrations, not observations.

Index Terms—*Distributed cloud, reliability, availability, common-cause failure, retry amplification, error budget, recovery time objective, recovery point objective*

1. Introduction
Cloud services are expected to be available on demand [1], and distributing them across zones, regions and providers is the usual way to meet that expectation. Reliability, however, is a property of the complete user journey rather than of individual components [2]. Studies of public cloud outages show that failures frequently arise from shared dependencies, configuration changes and recovery problems that defeat apparently redundant designs [3]. Partial or “gray” failures, in which components appear healthy to monitors while users experience errors, add further uncertainty [4].
Availability is therefore often over-stated when it is computed from component figures under unexamined independence assumptions. This paper, derived from a doctoral thesis, presents a quantitative but transparent assessment method in which every input is labelled as an assumption and every result is accompanied by its conditions. Its contributions are:

1) A six-step assessment method from service boundary to verification with measured inputs.
2) Sensitivity analyses of serial dependencies and of redundancy under common-cause failure.
3) Models of retry amplification and of error budgets that link client behaviour and objectives to load and tolerance.
4) A decomposition of recovery time and data-loss exposure that identifies where improvement effort has most effect.

2. Background
Distributed systems must cope with partial failure and uncertain communication. Under network partitions, a replicated service cannot simultaneously guarantee consistency and availability [5], and event ordering depends on logical relationships rather than wall-clock time [6]. Reliability therefore depends on application behaviour during failure, not only on infrastructure redundancy. At scale, rare slow or failed components affect a large share of requests, which motivates bounded, tail-tolerant request handling [7]. Chaos engineering tests resilience by deliberately injecting faults [8], and cyber-resilience guidance frames reliability as the ability to anticipate, withstand, recover from and adapt to adverse conditions [9].

ISSN: **3139-0862**

3. Assessment Method

Fig. 1 shows the method. The boundary step defines a user journey and the dependencies without which it cannot complete. The model step expresses those dependencies as serial, parallel and common-cause structures. The budget step states the objective and its measurement window. The stress step examines behaviour under retries, overload and failover, and the recovery step decomposes recovery time and data-loss exposure. Finally, every assumed input is replaced with measured data before any value is used for a decision.

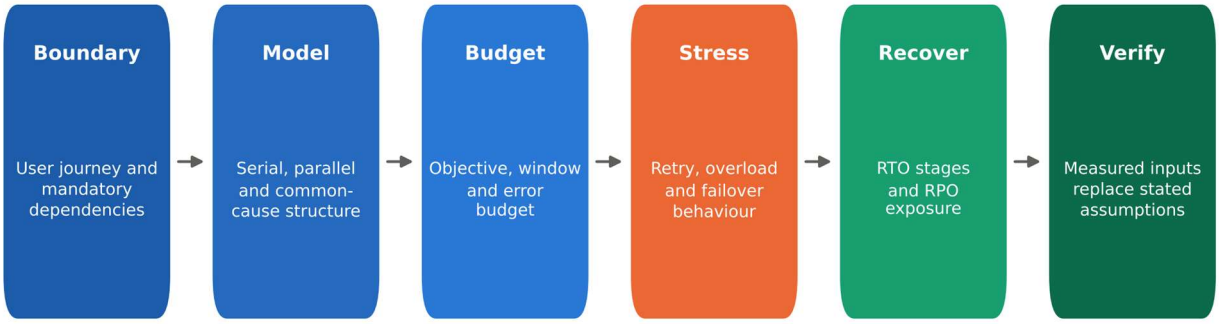

Fig. 1. Six-step reliability assessment method. Each step produces a calculation with stated assumptions that the final step replaces with measured inputs.

Table I lists the assumptions used in this paper. They are taken from the thesis's worked examples or chosen to show sensitivity; none is a measured value.

TABLE I STATED ASSUMPTIONS USED IN THE ANALYSES

Analysis	Assumed inputs	Main simplification
Serial dependencies	1–20 dependencies at 99.5%, 99.9% or 99.99%	Independent failures; same measurement window
Parallel redundancy	Two paths, each failing 0.1% or 1% of the time	Routing and capacity work during failover
Common cause	Shared failure probability 10^{-6} to 3×10^{-3}	Common cause disables both paths
Error budget	30-day window; objectives 99% to 99.99%	Time-based rather than request-based
Retry amplification	1–5 layers; 2–4 attempts per layer	Independent failure at each attempt
Recovery time	Stages of 2, 3, 1, 8 and 4 minutes	Stages are sequential
Data-loss window	10–1,000 acknowledged writes per second	Constant arrival rate; no durable replay

4. Availability Under Dependency Structure

4.1 Serial Dependencies

When a journey requires n components with availabilities A_i, all of which must succeed, the journey availability under independence is

$$A_s = \Pi_{i=1}^{n} A_i \qquad (1)$$

With three dependencies at 99.9%, Eq. (1) gives 99.7003%. Fig. 2(a) shows how this falls as dependencies are added: ten give 99.00% and twenty give 98.02%. At 99.5% per dependency, twenty give only 90.5%. Adding mandatory components such as identity, name resolution, configuration and feature-flag services therefore erodes availability even when each is individually reliable.

4.2 Parallel Redundancy and Common Cause

If either of two paths can serve a request and their failures are independent, with failure probability f each, the joint unavailability is f^2. When a common cause with probability c disables both paths,

$$U = 1 - (1 - c)(1 - f^2) \qquad (2)$$

With $f = 0.001$, the ideal parallel availability is 99.9999%, or 0.04 minutes of unavailability in 30 days. Adding the thesis's assumed common cause of $c = 0.0005$ lowers availability to 99.9499% and raises unavailability to 21.6 minutes (Fig. 2(b)). Above about $c = 10^{-5}$, the common cause rather than the paths determines availability. Shared identity, routing, deployment pipelines and configuration are typical common causes in cloud outages [3], so measuring and reducing them is more valuable than adding further replicas.

ISSN: **3139-0862**

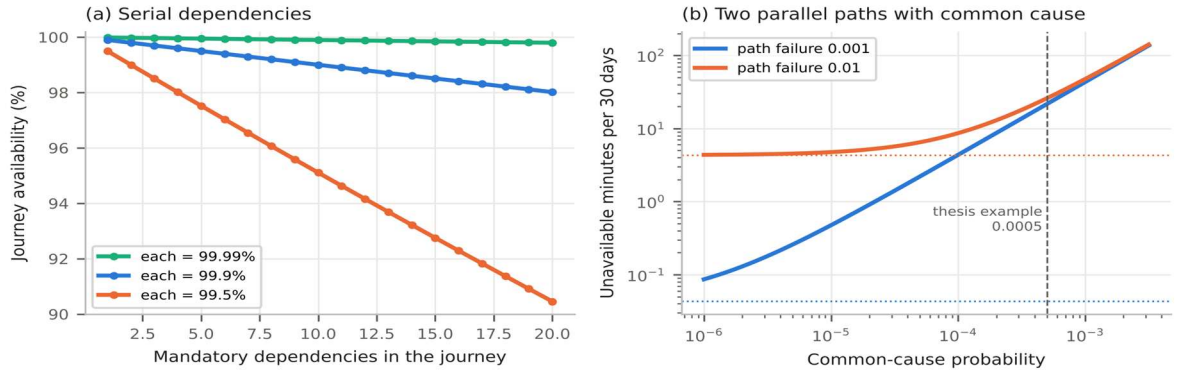

Fig. 2. (a) Journey availability as a function of the number of mandatory dependencies (Eq. 1). (b) Expected unavailable minutes per 30 days for two parallel paths as a function of common-cause probability; dotted lines show the ideal independent values (Eq. 2). Calculated illustrations.

5. Error Budgets and Retry Behaviour

5.1 Error Budgets

An availability objective defines how much unsuccessful service is allowed within a window. For a time-based objective A over a window of T minutes, the error budget is $(1 - A)T$. Table II gives the values for a 30-day window of 43,200 minutes. At 99.9%, the budget is 43.2 minutes, so the common-cause scenario of Section 4.2 alone would consume half of it. For request-based objectives, the budget is a share of eligible requests, and uneven traffic can make time-based and request-based interpretations diverge, so the denominator must be stated.

TABLE II ERROR BUDGETS FOR A 30-DAY WINDOW

Objective	Allowed unavailability	Minutes per 30 days	Share used by common-cause example
99%	1%	432.0	5%
99.9%	0.1%	43.2	50%
99.95%	0.05%	21.6	100%
99.99%	0.01%	4.32	Exceeded (500%)

5.2 Retry Amplification

Retries recover from transient faults but add load. If L layers each make up to a attempts, the worst-case number of attempts per original request is a^L. If each attempt fails independently with probability p, the expected attempts at one layer are

$$E_1 = \Sigma_{i=0}^{a-1} p^i, \quad E_L = (E_1)^L \qquad (3)$$

Fig. 3 shows both measures. Three layers with three attempts each give a worst case of 27 attempts per request, or 2,700 attempts per second for 100 original requests per second. The expected multiplier is modest when failures are rare—1.37 at $p = 0.1$—but reaches 5.4 at $p = 0.5$ and 19.9 at $p = 0.9$. Load therefore rises sharply exactly when a dependency is struggling, which can turn a partial failure into a cascade. A request-wide retry budget, for example limiting retries to 10% of traffic, keeps the multiplier close to 1.1 regardless of failure rate.

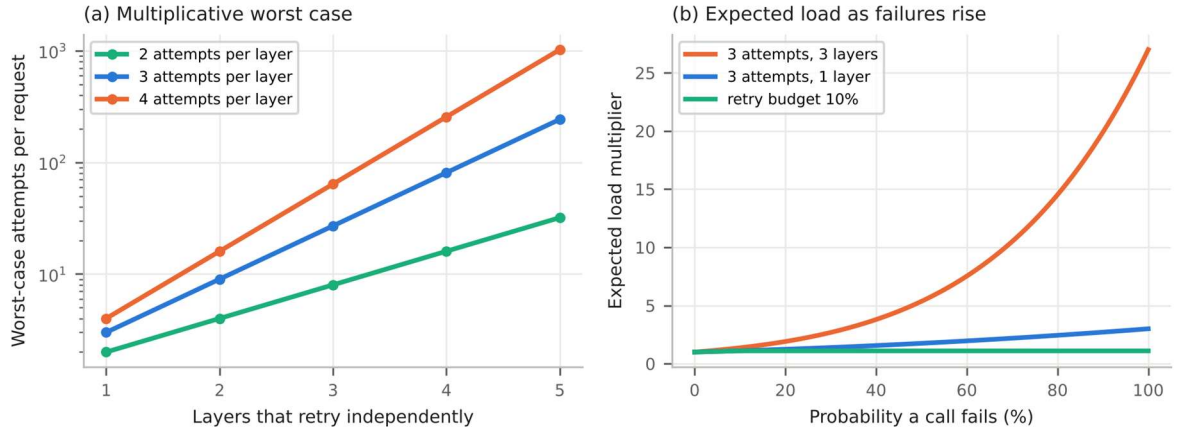

Fig. 3. (a) Worst-case attempts per request as a function of retrying layers and attempts per layer. (b) Expected load multiplier as the probability of call failure rises, with and without a 10% retry budget (Eq. 3). Calculated illustrations.

ISSN: **3139-0862**

6. Overload and Failover Controls

6.1 Timeouts and Deadlines

A timeout limits one call, whereas a deadline limits the whole request. With fixed per-hop timeouts, the worst-case wait grows with the number of hops and is multiplied by retries: four sequential hops with 500-ms timeouts can wait 2 seconds, or 6 seconds with three attempts per hop, although the user may abandon the request after 1 second (Fig. 4(a)). Propagating the remaining deadline lets each service avoid starting work that cannot finish in time and cancel work that is no longer needed. Cancellation must be handled carefully when side effects have already occurred elsewhere.

6.2 Circuit Breakers and Bulkheads

A circuit breaker stops repeated calls to a failing dependency, and a bulkhead limits how much of a shared resource one dependency can consume. In the example of Fig. 4(b), a hung dependency in a shared pool of 100 connections can occupy all of them, whereas a bulkhead that caps it at 20% leaves 80 connections for other work. Isolation must be tested at the real resource boundary: separate logical queues give little protection if their workers share one exhausted connection pool, and an unstable breaker can repeatedly interrupt a recovering dependency.

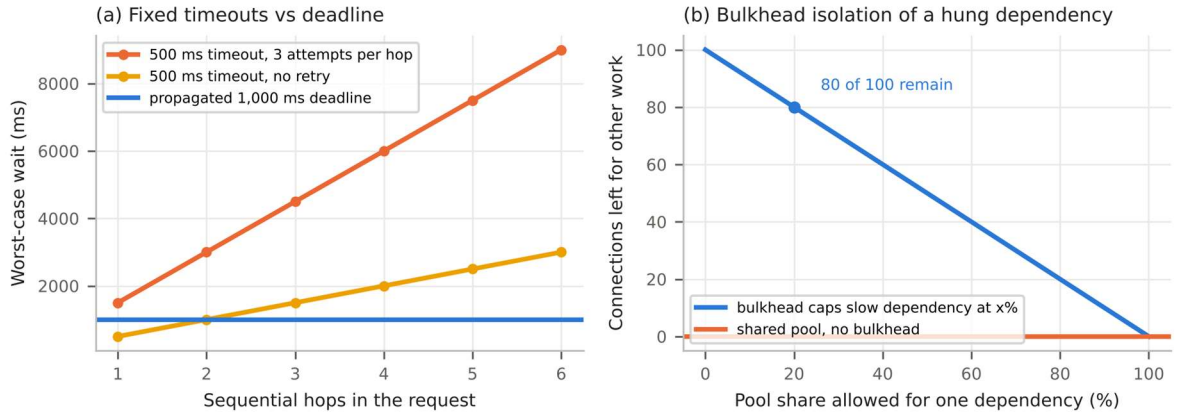

Fig. 4. (a) Worst-case waiting time with fixed per-hop timeouts compared with a propagated end-to-end deadline. (b) Connections left for unaffected work when one dependency hangs, with and without a bulkhead. Calculated illustrations.

6.3 Failover Correctness

Failover is correct only if the new target has acceptable data, sufficient capacity and valid authority to act. During a partition, each side may see the other as unavailable, so promotion based on reachability alone can create two writers [5]. Table III lists the checks that distinguish a safe failover from a simple health-based switch; these checks can make safe failover slower, which should be reflected in recovery-time objectives.

TABLE III FAILOVER CORRECTNESS CHECKS

Check	Question answered	Evidence
Data position	Is the target's data recent enough for the operation?	Replication position compared with recovery-point objective
Capacity	Can the target carry the redirected load?	Load test or reserved-capacity record
Authority	Is the target permitted to accept writes?	Coordination or lease record
Fencing	Can the former primary still write?	Fencing confirmation before promotion
Dependencies	Are identity, keys and configuration reachable from the target?	Dependency test from the recovery location

7. Recovery Time and Data Loss

7.1 Recovery-Time Decomposition

Recovery time consists of detection, diagnosis, decision, restoration and validation. With sequential stages of 2, 3, 1, 8 and 4 minutes, recovery takes 18 minutes (Fig. 5(a)). Restoration is the largest single stage, but halving it saves only 4 minutes, giving 14. Halving the other four stages saves 5 minutes, giving 13. Measuring only the restore command therefore understates recovery time and can misdirect investment; faster detection, clearer runbooks and pre-agreed decision authority can matter as much as faster infrastructure. Real exercises should record overlaps between stages and use an agreed validation endpoint.

7.2 Data-Loss Exposure

The acknowledged updates that may be lost depend on where acknowledgement occurs and which copies survive. Under a constant rate r of acknowledged writes and an unrecoverable interval t, the number of writes at risk is $r\,t$. At 100 writes per second, a 20-second interval

ISSN: **3139-0862**

exposes 2,000 writes (Fig. 5(b)). This is a count of potentially missing operations, not a measured loss: durable replay elsewhere can reduce it and bursts can increase it. A recovery exercise should reconcile individual operation identifiers against surviving durable records rather than rely on replication lag alone.

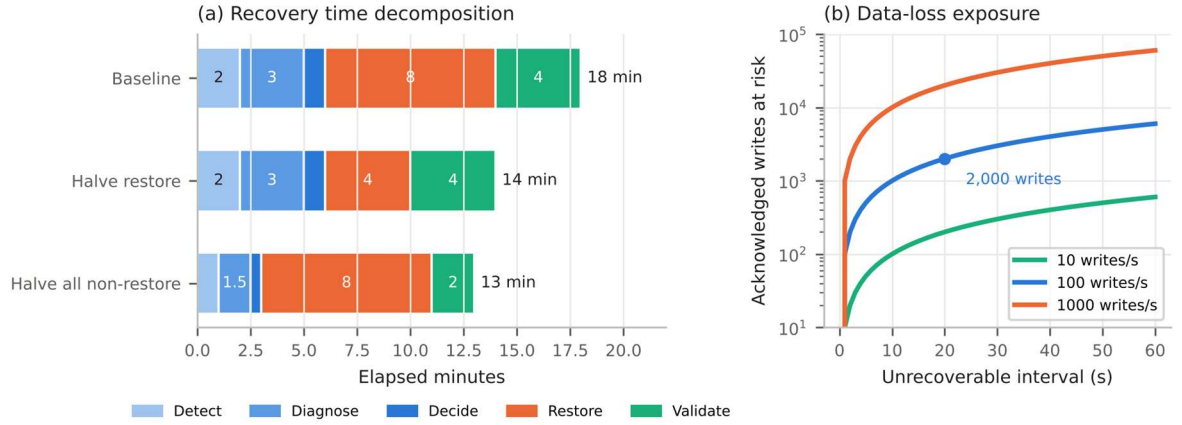

Fig. 5. (a) Sequential recovery-time decomposition for the baseline and two improvement options. (b) Acknowledged writes at risk as a function of unrecoverable interval and write rate. Calculated illustrations.

8. Integrated Worked Example

To show how the analyses combine, consider an illustrative order-placement journey with a 99.9% monthly objective (43.2-minute budget). The journey needs an identity service, an API gateway, an order service, a payment adapter and a database, each assumed to be 99.95% available, so the serial estimate is $0.9995^5 \approx 99.75\%$, already below the objective. Running the order service and database in two zones with independent path failure of 0.001 removes them from the serial chain, but a shared deployment pipeline acting as a common cause with probability 0.0005 restores much of the loss. Table V traces the effect of each design step under these assumptions.

TABLE IV ILLUSTRATIVE EFFECT OF DESIGN STEPS ON AN ORDER-PLACEMENT JOURNEY

Design step	Journey availability	Expected minutes lost per 30 days	Budget used
Single zone, five serial dependencies	99.75%	108	250%
Order service and database in two zones (independent)	99.85%	65	150%
Same, with a shared-pipeline common cause (0.0005)	99.80%	86	200%
Separate pipelines and staged release (common cause 0.00005)	99.85%	67	155%
Also: identity and gateway made redundant across zones	99.94%	24	55%

The example shows that redundancy helps only after the shared causes have been addressed, and that the remaining serial dependencies must also be addressed: making identity and gateway redundant brings the journey within its budget, with the single payment adapter now the largest contributor. It also shows why a single availability figure is insufficient: each row depends on stated assumptions that must be measured before the design is accepted. Retry budgets and propagated deadlines (Section 6) then determine whether a partial failure of one dependency stays within this budget or cascades into the others.

9. Discussion

9.1 Implications for Design

Four implications follow. First, every mandatory dependency added to a journey reduces availability, so dependency inventories should precede availability claims. Second, common causes dominate redundant designs, and the independence of replicas must be demonstrated rather than assumed. Third, retry policies should be coordinated across layers with a request-wide budget, because uncoordinated retries amplify load during partial failures. Fourth, recovery improvement should target the whole recovery sequence, including detection, diagnosis and decision.

ISSN: **3139-0862**

9.2 From Calculation to Evidence

The analyses are calculated illustrations, so they show the shape of each trade-off rather than its size in any particular system. Table IV lists the measurements needed to replace each assumption. Fault-injection experiments [8] and recorded recovery exercises are the main sources of such evidence.

TABLE V MEASUREMENTS REQUIRED TO REPLACE ASSUMED INPUTS

Assumption	Measurement needed	Typical source
Component availability	Success rate per dependency on the user journey	Request-level telemetry
Independence of paths	Shared identity, routing, release and configuration paths	Dependency inventory; outage reviews
Retry behaviour	Attempts per request by layer during faults	Distributed tracing under fault injection
Recovery stages	Timestamps for detection, decision, restoration, validation	Recovery exercise records
Data-loss window	Acknowledged operations reconciled with surviving state	Commit logs and operation identifiers

9.3 Limitations

The models assume independence where stated, constant rates and sequential recovery stages, and they omit correctness: a path that responds but returns wrong or stale data is not available in the user's sense [4]. Cost and staffing effects are outside the scope of this paper.

10. Conclusion and Future Work

This paper presented a transparent method for quantitative reliability assessment of distributed cloud services and applied it to serial dependencies, common-cause failure, error budgets, retry amplification, recovery time and data-loss exposure. The calculations showed that a 0.05% common cause can raise expected unavailability of an ideal redundant pair from 0.04 to 21.6 minutes per month, that uncoordinated retries can multiply load by 27 in the worst case, that non-restoration stages can account for more than half of recovery time, and that propagated deadlines and bulkheads bound the effect of slow dependencies. Future work will replace the assumed inputs with measurements from fault-injection experiments and recovery exercises on deployed services.

REFERENCES

[1] P. Mell and T. Grance, "The NIST definition of cloud computing," National Institute of Standards and Technology, Gaithersburg, MD, USA, NIST SP 800-145, 2011, doi: 10.6028/NIST.SP.800-145.

[2] A. Avižienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Trans. Dependable Secure Comput.*, vol. 1, no. 1, pp. 11–33, 2004, doi: 10.1109/TDSC.2004.2.

[3] H. S. Gunawi *et al.*, "Why does the cloud stop computing? Lessons from hundreds of service outages," in *Proc. 7th ACM Symp. Cloud Comput. (SoCC)*, 2016, pp. 1–16, doi: 10.1145/2987550.2987583.

[4] P. Huang *et al.*, "Gray failure: The Achilles' heel of cloud-scale systems," in *Proc. 16th Workshop Hot Topics Oper. Syst. (HotOS)*, 2017, pp. 150–155, doi: 10.1145/3102980.3103005.

[5] S. Gilbert and N. Lynch, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *ACM SIGACT News*, vol. 33, no. 2, pp. 51–59, 2002, doi: 10.1145/564585.564601.

[6] L. Lamport, "Time, clocks, and the ordering of events in a distributed system," *Commun. ACM*, vol. 21, no. 7, pp. 558–565, 1978, doi: 10.1145/359545.359563.

[7] J. Dean and L. A. Barroso, "The tail at scale," *Commun. ACM*, vol. 56, no. 2, pp. 74–80, 2013, doi: 10.1145/2408776.2408794.

[8] A. Basiri *et al.*, "Chaos engineering," *IEEE Softw.*, vol. 33, no. 3, pp. 35–41, 2016, doi: 10.1109/MS.2016.60.

[9] R. Ross *et al.*, "Developing cyber-resilient systems: A systems security engineering approach," National Institute of Standards and Technology, Gaithersburg, MD, USA, NIST SP 800-160 Vol. 2 Rev. 1, 2021, doi: 10.6028/NIST.SP.800-160v2r1.