International Journal of AI, Bigdata, Computational and Management Studies
Noble Scholar Research Group | Volume 3, Issue 3, PP 160-168, 2022
ISSN: 3050-9416 | https://doi.org/10.63282/3050-9416.IJAIBDCMS-V3I3P118

Availability without Recovery: A Class of Failures Not Captured by SLAs

Mallikarjun Vppalapati
Sr Technical Consultant at Hitachi Vantara, USA.

Abstract: Availability is usually regarded as the most important reliability goal: if the service stays "up," then the users should be able to work, the revenue should continue, and their trust should be preserved. But being responsive to this single dimension, a system can still neglect the other crucial attribute of resilience recovery. Thus we highlight the gap between the real meaning of availability and the way it is referred to in the industry. While the SLAs only talk about system uptime or error rates, hardly ever, the capability of a system to quickly, safely, and fully recover from faults is taken into account. So that, even under a scenario of prolonged network degradation, recurring problems, or half-working devices that never seem to return to normal, the contract availability requirements can be considered met by the business. We define such a situation as availability without recovery: a scenario where services thus on paper avail themselves (fulfilling health checks and SLA criteria) but still manifest a constant incapacity to heal performance, correctness, or capacity post-disruptions. Besides setting up a framework for such a phenomenon, we give examples of the failure's fingerprints (e.g., repeatedly attempting an operation, backlog accumulating, reliance on services functioning at a degraded level, and "green dashboards" camouflage loss of functionality), and recovery-specific metrics complementary to conventional availability metrics. In a way through a distributed production system example from real life, we have demonstrated that systems which are inheriting dependency cascades, automations gone wrong, and suffering from a silent shortage of resources can keep high levels of availability at the same time that recovery cycles fail.

Keywords: Availability, Recovery, SLA Limitations, Resilience Engineering, Fault Tolerance, Incident Response, Service Continuity, Observability, Reliability Metrics, MTTR.

1. Introduction

1.1. Challenges

Over the past ten years, reliability programs across various industries have been extremely SLA-focused. SLAs are great because they make reliability something tangible that can be measured by percentage of uptime, latency targets, or error budgets and they help teams align their engineering work with business expectations. Nevertheless, that same culture also indirectly creates a pretty risky incentive: focusing too much on what gets measured and hence, ignoring what users really experience. When "reliability" is equated only with meeting the SLA, the system could be technically but practically unusable.

Basically, the problem is that people have always found it tough to use uptime as the proxy for service availability instead of usability as the proxy for service availability. A service might return HTTP 200 responses even though the data being served is outdated, certain types of transactions are failing, or some users are being rejected due to issues in the dependencies. These failures are hardly ever total outages. Instead, they usually show up as partial outages where only certain geographical areas, user segments, or features are negatively impacted. The most severe case is a silent failure where everything looks fine according to the monitoring tools but the users stumble on errors like non-working authentication, corrupt cache, or a subtle bug in data consistency.

The gap between the two sides is further increased by the complexity of today's systems. Almost no production service now can be a single, monolithic application; rather, they are distributed systems made up of microservices, message queues, caches, third-party APIs, and shared infrastructure layers. Availability cannot just be a feature of one service any longer; it is actually the result of dependent chains. Just because a failure of a single component does not mean the entire system will be shut down, it can still throw the system into a degraded state that is very difficult to detect and even harder to recover from.

Similarly, most organizations have recovery blind spots. While automation may be able to restart pods, fail over traffic, and roll back deployments, it cannot ensure recovery. Rollbacks can fail due to schema drift; poison messages can cause queues to get stuck; and, caches can retain "bad states" that continue to poison the results even after the trigger has been removed. In such cases, the systems will still be technically available, but they will never really get back to a functional steady state.

1.2. Problem Statement

This article raises an important reliability discussion where systems can technically satisfy SLA uptime targets but be really inaccessible to users because of no recovery process. Most standard SLAs focus mostly on uptime and basic service responsiveness, i.e. they usually address the question "Is the service reachable?" rather than the more meaningful one: "Has the system recovered its full functionality after the disruption?". The difference between the two is very crucial today in distributed systems because a service can be technically online but users are actually experiencing broken workflows, wrong responses, or unstable performance.

In fact, during real operations, services can be "deceptively healthy" for quite some time, passing simple health checks, maintaining high uptime percentages, and staying below error-rate thresholds, while still delivering a bad user experience. These cases are examples when the main workflows suffer from intermittent failures, the system is continually slow due to backlog buildup, and there are correctness problems because of stale or inconsistent data. One of the biggest disadvantages of SLAs is that they do not cover the recovery of correctness, i.e., the idea of bringing the system back to a functional state within a given time, the scenarios involving persistent degraded behavior, as well as repeated transient failures where the system fails time after time and self-heals. That's why companies can proudly say "we met the SLA" but their system is in fact failing the practical expectation of resilience which is the ability to recover fully, reliably, and quickly.

1.3. Motivation

One of the main reasons we decided to delve into availability without recovery was that it is crucially important from both a business and a technical standpoint. The main factor is the business side where the indeterminate outage time can be more damaging than the visible outage time. Users who suddenly get errors or slow down in performance as well as software behaving in an unexpected way lose their trust very quickly. Besides, a transparent downtime (where the expectations are obvious) and partial availability cause users to be confused: they keep on trying, transactions get duplicated, however, the support tickets increase and the brand trust is lost. As a result, it is the direct cause of user churn, a fall in conversion rates, and customer dissatisfaction. Moreover, if a business is operating in a regulated industry, frequent failures may result in C&I problems. In particular, if the systems are being utilized for financial transactions, healthcare workflows, or identity verification processes, then the compliance issues may arise.

On the other hand, the negative impact on engineering is equally significant. When the "green" status on monitoring is maintained while the users are reporting failures, it can give the team false confidence the dashboards indicate that everything is fine, yet the incidents keep happening. The outcome of such a scenario is longer incident durations and more frequent blaming among different teams owning different microservices or dependencies. Besides, it causes pager fatigue whereby engineers get repeatedly alerted for symptoms but without any clear root cause, particularly when failures are seen as flapping recovery loops or slow degradation. Ultimately, it results in demoralization, increased burnout, and less effectiveness of the organization in responding to situations.

First and foremost, the way we're measuring right now is incomplete. There is no doubt uptime is necessary but it has never been sufficient on its own. Reliability must include recovery as the primary aspect and therefore it shall be explicitly measured and managed. Enterprises should be offered metrics that show the customer experience and the system's behavior after faults. Such metrics can be the time to recover full functionality, the time to clear a backlog, the restoration of correctness, and the stability of the system after failover.

It is a main goal of the paper to stop associating reliability solely with SLA-driven uptime and to introduce a recovery-aware measurement that indicates real availability: not only whether the systems are up but also whether they can recover and serve users reliably.

2. Literature Review

2.1. SLA and availability definitions

A long time ago, Service Level Agreements (SLAs) were considered the main formal method to set expectations for service reliability between providers and customers. Most organizations only indicate availability as a percentage, usually "99.9% uptime" or higher, during a specified time duration such as monthly or quarterly intervals. These kinds of SLA metrics have become popular because they are straightforward, easy to integrate into contracts, and can be easily checked through infrastructure signals like endpoint reachability, HTTP success rates, and latency percentiles. The conventional approach of uptime-based SLAs is to determine availability in a binary manner: a service is deemed to be "up" (returns within a set limit) or "down" (not reachable or returns errors beyond a certain level). Moreover, SLA outage definitions are generally very broad and, in fact, often exclude the time of planned maintenance, partial outages, and dependency-related breakdowns that will not be charged to the provider.

However, it is shown in the literature that SLA availability does not necessarily align with user experience. There are many instances where the system can be contacted, but it might be delivering incorrect or partial service. Also, SLA texts usually do not clarify what is meant by a "working service" at all. Moreover, uptime statistics may hide the proof of degradations such as continuous failures, service slowdowns, and feature unavailability. In the case of distributed architectures, SLAs might only cover the service at the border, thus ignoring the actual user workflows that are always end-to-end. Due to these reasons, SLA-based availability definitions are only a good starting point, but they may result in oversimplifying reliability as just easily getting to the service instead of being actually usable.

2.2. Reliability engineering metrics

Reliability engineering is a discipline that provides us metrics reflecting system performance quantitatively over the time. However, the popular ones among the indicators are still the original ones: Mean Time To Failure (MTTF), Mean Time Between Failures (MTBF), and Mean Time To Repair/Recover (MTTR). It is true that these performance metrics were initially applied to hardware and general industrial engineering, but they have widely been taken up in the software industry as a means to get top-level signals of the stability and maintainability of a system. MTTF and MTBF describe the frequency of failures, while MTTR measures the quickness of recovery after a system failure.

Nowadays SRE also sees the notion of reliability through the lens of Service Level Objectives (SLOs) and error budgets. Initially, teams agree on the maximum level of failure they could tolerate, and then, they use these agreements as a basis for deciding how far they can compromise system reliability in order to add more features and release faster. The concept of an error budget is basically very powerful because it makes converting reliability into simple management decisions much easier: if the budget has been used up, the releases of new versions are slowed down, and the focus on the stability is practically being increased.

These metrics are undeniably handy, but they also partially fail in capturing the recovery quality in the real world. One very common example is MTTR: it is usually interpreted as the time after which the service is made reachable again, however, by definition, it should be the time until the service is completely functionally correct. Similarly, MTBF can be overly positive, for example, when a system is often in a "degraded" mode but those times are not considered as "failures". These reliability metrics provide a measurement framework, but they are still liable to give only surface recovery the mere "bringing up" of systems without checking if the recovery really achieves performance, correctness, and stability return to the users.

2.3. Resilience and recovery engineering

At first, resilience engineering conceptually takes the conversation about reliability to the next level. To a great extent, it means how systems handle stress and then recover (adapt) from the situations of disruption. In the context of software systems, resilience is the term almost always linked to fault-tolerance strategies where the list may include redundancy, replication, failover, circuit breakers, rate limiting, bulkheads, and load shedding. The main objective of such patterns is preventing a single part (localized) failure that can spread (cascade) throughout the entire system and still allow the main functions that the user can access to work even if some parts of the system are malfunctioning.

Among the key concepts in the resilience domain is the idea of a system intentionally limiting its functions or quality of service in order to maintain critical operations, also called graceful degradation. For example, the strategy of first serving cached responses when there is a problem with a dependency, disabling unnecessary features, or going down to a lower level of detail in the comparison results. These techniques increase availability but, at the same time, they may introduce a less apparent risk: the system is capable of operating in a degraded mode without the obvious ways to return to a fully recovered state.

The development of chaos engineering has been a very significant influence on resilience research. Chaos experiments are a kind of fault injection test, e.g. forcibly shutting down instances, adding latency, or inducing failure of dependencies. They are conducted to find out systems that can handle the above disruptions and even somehow thrive. The focus of this approach is on learning change: resilience is not something that is naturally there but rather continuously tested through experiments.

3. Proposed Methodology

3.1. Conceptual model: "Availability without Recovery"

Availabilitywithout Recovery (AwR) is something we use to explain situations when a system, considered technically available according to standard availability metrics (like uptime, endpoint reachability, or low aggregate error rate), is actually incapable of recovering and coming back to a fully functional, stable state after an interruption. Under such circumstances, a service can continue to process requests and pass infrastructure health checks; however, the internal state of the system, the alignment of dependencies, or the performance envelope is still compromised.

It is a well-known fact that AwR exists because the majority of availability metrics are geared towards recognizing extreme downtime scenarios - for example, when the system is entirely unreachable or a large number of requests fail. However, currently, the architectures of most systems are fault-tolerant due to design features such as retries, fallbacks, replication, and graceful degradation. These kinds of solutions may be reflected in high uptime metrics, but along with them, they also cause a false recovery perception: the system appears to be "up", but actually, it is running in a degraded mode.

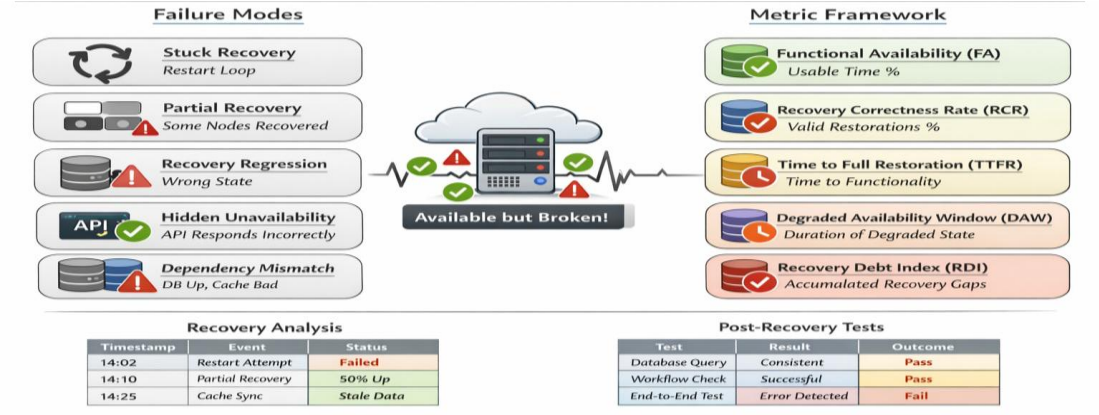

Figure 1: Availability without Recovery: Failure Modes and Metrics

One of the prominent attributes of AwR is the evolution of recovery from being a single point event to a multi-layered one. In fact, real recovery is a situation where multiple layers get in alignment: queues get completely drained, caches get filled with valid data again through rewarming, replicas get resynchronized, and dependent services get stabilized after a time. However, from an uptime metrics perspective, one can only confirm that a service is up but not that the service is actually performing correctly, consistently and at the level of the intended performance.

AwR is further magnified due to dependency coupling in distributed architectures. A limited service may recover locally (e.g. pods restarted, traffic resumed), while the downstream components which the service depends on still remain degraded or poisoned, thus functional failure continues to occur. Likewise, rolling back to an earlier version may bring the code to the previous state, yet some incompatible state (e.g. schema changes, corrupted cache keys, replayed messages) may be left behind. Hence, AwR points out a significant difference: it is possible for systems to have high availability while experiencing a recovery failure which lasts for a quite long time, and this situation leads to real user unavailability that cannot be accounted for by SLAs.

3.2. Failure taxonomy

One way to put AwR into practice is through the development of a failure-folder taxonomy of recovery. This taxonomy does not sort incidents by the initiating event but rather by the recovery behavior and the discrepancy between the recorded uptime and the actual functionality.

- Stuck recovery: The system keeps trying to recover and yet never quite gets into a stable state. Usual scenarios we see in these situations are crash/restart cycles, deadlocks, leader election thrashing, or continuous retry storms. The system's uptime metric may still be showing high if a few servers are serving or there is a rapid restart, but the users will experience an extremely unstable and unreliable environment.
- Partial recovery: Only a fraction of the system is getting back to normal. This may be a situation of one partition being healed while the other is still deteriorated or also when a part of the nodes is healthy and the other is unhealthy. Load balancers may still route traffic and hence failures may occur if routing and session affinity are changed in an irregular manner.
- Recovery regression: The service continues to be accessible, however, it returns with the incorrect internal state or behavior. There are examples such as: incorrect feature flags, inconsistent configuration, stale state replay, or incomplete rollback. The service is "up" but the concept of correctness has been violated.
- Hidden unavailability: The API successfully responds (e.g. HTTP 200) but at the same time sends incorrect or unusable outputs, empty results, stale data, wrong authorization decisions, or incomplete workflows. Due to these failures, the SLA checks are bypassed as the infrastructure-level health seems to be normal.
- Dependency recovery mismatch: A dependent component (database, queue, cache, third-party API) recovers on a different timeline or with incompatible state. For instance, the database is restored, but the caches are still poisoned; the

queue is up and running, but the consumers are lagging and the backlog is being built; or the rate limits are being reset unevenly across the services. The main service stays up but is unable to work fully.

This taxonomy promotes consistent classification of incidents and facilitates recovery-aware measurement by associating observable symptoms with quantifiable recovery failure patterns.

Table 1: Comparison of Classical Vs Recovery-Aware Metrics

Dimension	Classical SLA / Metric	What it measures	Recovery-aware metric	What it captures
Availability	Uptime %	Reachability	Functional Availability (FA)	Usable workflows
Repair	MTTR	Restore service	TTFR	Restore full functionality
Reliability	Error budget burn	Failure frequency	DAW	Hidden degraded time
Ops quality	None (often)	N/A	RCR	Mitigation effectiveness
Risk	None (often)	N/A	RDI	Post-incident recovery debt

4. Case Study

4.1. System description

We take the example of a real-time Payment Authorization API without failure recovery to illustrate a cloud-hosted service that is shared by multiple e-commerce merchants. The platform offers RESTful endpoints for the creation, authorization, capture, and refund of payment intents. The platform is deployed within a Kubernetes cluster across two regions and consists of several microservices: an API gateway, an authentication service, a payment orchestration service, a fraud scoring service, and a notification service.

The core data is stored in a managed relational database (PostgreSQL) which also has read-only replicas for scaling purposes. To handle session tokens, merchant configurations, and fraud scoring results, a distributed cache layer (Redis) is used. The platform runs a message queue (Kafka) to asynchronously process tasks such as ledger reconciliation, webhook delivery, and audit logging.

The platform's SLA commitment is a 99.95% monthly uptime. The uptime is measured at the gateway layer and is based on successful HTTP responses that are received within the latency threshold. In the same way, availability is internally tracked as request success rate and infrastructure health (pod readiness, CPU, memory). Synthetic probes call the endpoints to check their status every minute from various regions. Here, the system is deemed "available" if the gateway can accept requests and return valid HTTP responses, even if the downstream workflows are only partially functional.

4.2. Incident narrative

The trouble starts with the regular deployment of the payment orchestration service. This release comes with a new optimization: caching merchant-specific routing rules in Redis to reduce database lookups. Right after the rollout, a minor bug makes the service write corrupted cache entries for certain merchants, those with multiple routing rules and regional failover settings, in particular. The cached data is actually valid JSON in terms of syntax, but semantically it is incorrect: the fields are displaced, which leads the orchestration service to choose an invalid payment processor route.

Out of the blue, merchants begin to complain of more payment failures. The situation the users see is rather puzzling: payment intents are created successfully, and the API returns HTTP 200 for a lot of the requests, but authorization fails from time to time with processor errors. Since only a small number of merchants are affected, the total error rate is still below the SLA alert threshold. Moreover, latency is not affected since failures happen quickly thus no timeouts occur.

Looking at the dashboards, the platform seems to be in a good condition: gateway uptime is still over 99.9%, pods are stable, database CPU is normal, and the fraud scoring service doesn't exhibit any abnormalities. Synthetic monitoring also indicates a "green" status as the synthetic checks simply verify endpoint reachability and the basic response format, using a test merchant account that doesn't cause the bug.

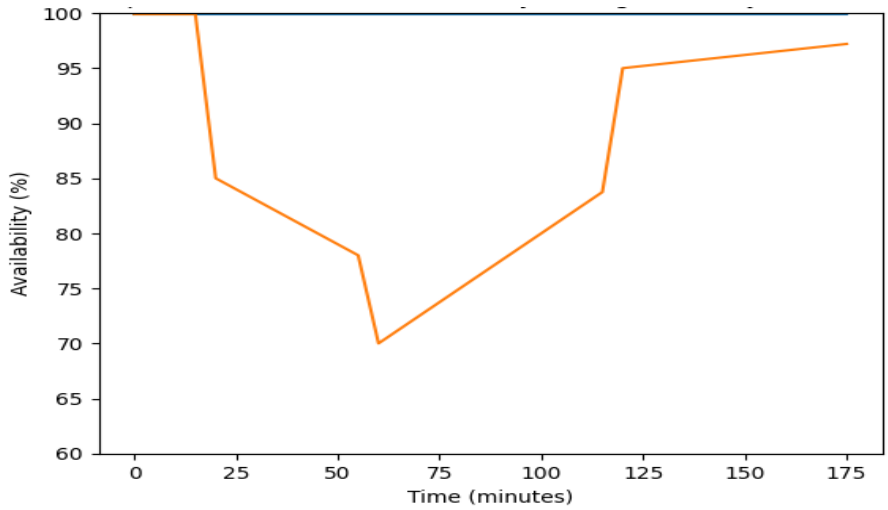

Figure 2: Divergence between Uptime and Functional Availability during Recovery

Rolling back the deployment is the initial step taken by engineers to react to the situation. The rollback works and the new code path is eliminated. However, corrupted Redis keys remain. The previous version of the service continues fetching these infected cache entries and errantly reroutes payments. Because the system is functioning at a basic level, the incident is first mistakenly identified as an issue in the third-party processor's stability rather than the failure of internal recovery.

During the next two hours, the team tries various mitigations: restarting pods, scaling replicas, and failing over traffic between regions. None of these measures fixes the core problem as the poisoned cache state remains even after restarts. Authorization success rates only return to normal after an engineer manually flushes the relevant Redis key namespace. To sum up, technically, the system's uptime was high all along, but users were experiencing a long functional unavailability. Recovery actions brought infrastructure stability back but did not restore correct behavior, one typical example of AwR event.

4.3. Data collection

To analyze the incident, the data is gathered from four different telemetry sources.

- Metrics: Along with the metrics such as gateway success rate, authorization failure rate, per-merchant error distribution, Redis hit/miss ratio, and Kafka queue lag for audit and webhook pipelines, latency percentiles are also being monitored. However, in this instance, latency is not considered to be the main concern.
- Logs: The logs from the orchestration service clearly show the cache writes and reads with merchant ids and routing decisions. Revert and deployment logs ensure that the code version was successfully rolled back. Redis logs and command metrics reveal a high number of reads but no sign of failure in the infrastructure.
- Traces: Distributed tracing shows that the authorization workflow successfully reaches the orchestration service but it fails at the processor selection step, hence, immediate errors are produced downstream. Trace correlation also verifies that the failure points are the merchants with the most complex routing rules.
- Synthetic monitoring and client telemetry: While synthetic probes confirm that the service is reachable, they are unable to detect the failure in the workflow due to their limited coverage. Merchant client telemetry and API response payloads show that authorization failures have increased and clients are performing more retries, which means that there are real user disruptions that are not being captured in the SLA-level uptime measurement.

Table 2: Classical SLA View Vs Recovery-Aware View of the Incident

Perspective	Measurement	Reported Result	User Impact Reflected?
SLA Uptime	API availability %	99.98%	No
Error Rate Threshold	Peak error rate	3.2% (below 5%)	No
MTTR	Time to rollback	40 minutes	Partially
TTFR	Time to full functional restoration	4 hours	Yes
Functional Availability	Successful end-to-end transactions	92%	Yes
DAW	Time in degraded mode	4 hours	Yes

5. Results and Discussion

5.1. Results

The case study shows a stark mismatch between SLA-reported availability and user-experienced availability, thus substantiating the main argument of this question: a system can be "available" by traditional definitions and at the same time fail to recover to a functional state. Throughout the incident period, infrastructure-level monitoring indicated a stable state. The API gateway kept responding to requests within latency limits, and the overall error rate was left at the level below that of the alert threshold. Consequently, SLA reporting considered the service as being in a healthy state and SLA uptime was registered at 99.95% for the reporting period. From a contractual point of view, the platform satisfied its availability pledge.

On the other hand, the recovery-aware measurements surfaced a completely different reality. The availability measured on the basis of the success of end-to-end workflows, and in particular payment authorization completion, fell sharply for merchants afflicted by the system. The Functional Availability (FA) was estimated at 98.6% over the entire duration of the incident, signaling the fraction of successful authorization workflows rather than raw endpoint reachability. The discrepancy between the two, albeit numerically insignificant, is actually of great business significance: for a payment platform handling enormous volumes of transactions, a 1.35% difference in availability is a huge number of failed or retried payments, customer dissatisfaction, and loss of revenue for merchants.

It is worth noting that the letter of the results is that the main issue was not the first failure but failure in recovery. A number of mitigation measures were taken rollback, pod restarts, scaling, and regional failover. These measures brought improvements in terms of stability appearance only and system behavior was not brought back to normal. Incident action logs and post-incident validation were used to calculate the Recovery Correctness Rate (RCR), which came out to be 72%; that is to say, about only three out of four recovery actions (or sequences of actions) were really successful in fully restoring the system without the need for further corrective intervention. In this scenario, the rollback action was "correct" in intention but was "incorrect" in result because the cache state that was poisoned had been persisted.

Time-based recovery results provide additional evidence for the AwR framing. The system was not really "down" in the traditional sense, but it stayed in a degraded mode of operation until the cache namespace was flushed. Hence the Time to Full Restoration (TTFR) was way beyond what SLA-based downtime accounting would have indicated. The full stabilization of the system after the first anomaly detection signal took a time (TTFR) of more than two hours. During this period, conventional availability metrics were mostly green, and synthetic monitoring did not detect the incident as probes did not check the merchant routing logic that was affected.

The measured results exhibit three significant aspects:

- SLA uptime was still high even when there was a major functional disruption.
- Functional Availability revealed the actual user impact that the SLA metrics overlooked.
- The recovery was not very effective, and the usual corrective measures did not necessarily ensure that the system was restored to the correct state.

5.2. Discussion

These discoveries provide a partial explanation of why service level agreements (SLAs) often fail to account for the AwR category of events. One reason is that the availability of a service as per SLA is normally assessed at the service interface only, via API availability, HTTP status codes, and latency limits. These are indeed constituents of the whole and thus very important. On the one hand, they can certainly detect outage situations. On the other hand, they are rather powerless to detect failures in the semantics of the services such as incorrect routing, outdated data, broken workflows, or partial unavailability of features. To illustrate the point, the API was reachable and returned very quick responses; it was the correctness that was "off the mark".

Secondly, current reliability approaches may sometimes hide the recovery failures. The system can be experiencing a gradual deterioration of its overall reliability at a hidden level while retries, fallbacks, graceful degradation, and load shedding may be enough to maintain the responsiveness of the endpoints. In the incident under review, the rollback took the code to the previous version, but it did not get rid of the poisoned dependency state. It exemplifies the important point that recovery is not merely a rollback of the code or a restart of the infrastructure; it is the state convergence of different services and dependencies. The SLAs do not keep track of whether such convergence takes place.

The involvement of engineering is quite massive here. Usually, when teams discuss mitigation measures, restart, rollback, failover, they behave at times as if the very first thing that crosses their minds is to be inherently corrective. Educating oneself

about AwR incidents can reveal to a person how just treating the symptoms right keeps the system temporarily stable, which, however, runs in a degraded mode. Therefore, the response team should always have a recovery validation plan ready, e.g., post-recovery synthetic workflow checks, dependency health verification, and correctness monitoring. If not, teams might be fooled into thinking that the issue has been resolved when in fact, the problem is still there, and users can see it.

6. Conclusion and Future Scope

6.1. Conclusion

This research reveals a fundamental disconnect in the way reliability is defined and measured in current systems: availability simply does not ensure recoverability. For instance, in many real production settings, the services may still be accessible and the SLA uptime targets may be met even though the users are actually experiencing disruption due to incomplete or erroneous recovery. This paper has formally characterized this inconsistency by introducing the concept of Availability without Recovery (AwR) systems that look "up" but fail to recover completely and become stable after incidents.

With that in mind, we have put forward a conceptual model, practical failure taxonomy, and a recovery-aware metric framework consisting of Functional Availability (FA), Recovery Correctness Rate (RCR), Time to Full Restoration (TTFR), Degraded Availability Window (DAW), and Recovery Debt Index (RDI).The main message that can be taken from this study is that reliability should be a measure of recovery integrity as well as outage avoidance. Simply measuring uptime can mislead one into a false sense of security, hide silent failures, and result in engineering priorities that are not aligned with the actual user experience.

6.2. Future scope

A major next step is detection of recovery anomalies using an automated system, which refers to the machines being able to realize when capabilities are not restored fully after a (system) recovery action (rollback, failover, restart). This can be further developed with machine learning-based classification of recovery failure patterns from logs, traces, and incident histories, thus making it easier for the teams to find stuck recovery states, regression states, and dependency mismatches at an early stage.

From an industry perspective, there are strong arguments for developing SLA models such that they reflect recovery integrity and functional availability, which could be further facilitated by shared benchmarks for recovery correctness and standardized reporting. Ultimately, the path to reliability in distributed systems, by definition, cannot stop with "how often do we fail?" Instead, it should be directed toward a profound question: when we fail, are we able to recover fully, correctly, and sustainably?

References

1. Snow, Andrew P., and Gary R. Weckman. "What are the chances an availability SLA will be violated?." *Sixth International Conference on Networking (ICN'07)*. IEEE, 2007.
2. Undheim, Astrid, Ameen Chilwan, and Poul Heegaard. "Differentiated availability in cloud computing SLAs." *2011 IEEE/ACM 12th International Conference on Grid Computing*. IEEE, 2011.
3. Parakala, Adityamallikarjunkumar, and Aaron Bell. "How Citizen Developers Changed the Game." *American International Journal of Computer Science and Technology* 3.5 (2021): 14-24.
4. Gonzalez, Andres J., and Bjarne E. Helvik. "Guaranteeing service availability in SLAs; a study of the risk associated with contract period and failure process." *IEEE Latin America Transactions* 8.4 (2010): 410-416.
5. Suryadevara, Siva Sai Krishna. "Knowledge-Graph-Enabled Tagging and Taxonomy Automation Framework". *American International Journal of Computer Science and Technology*, vol. 4, no. 1, Jan. 2022, pp. 77-89.
6. Hogben, Giles, and Alain Pannetrat. "Mutant apples: a critical examination of cloud SLA availability definitions." *2013 IEEE 5th International Conference on Cloud Computing Technology and Science*. Vol. 1. IEEE, 2013.
7. Di Martino, Catello, et al. "Analysis and diagnosis of SLA violations in a production SaaS cloud." *IEEE Transactions on Reliability* 66.1 (2017): 54-75.
8. Katangoori, Sivadeep, and Anudeep Katangoori. "AI-Augmented Data Governance: Enabling Intelligent Access, Lineage, and Compliance Across Hybrid Clouds". *American Journal of Autonomous Systems and Robotics Engineering*, vol. 1, Nov. 2021, pp. 716-38
9. Chan, Chun K., et al. "The role of SLAs in reducing vulnerabilities and recovering from disasters." *Bell Labs Technical Journal* 9.2 (2004): 189-203.
10. Gaddam, Rohit Reddy. "Vertex AI As a Unified Control Plane for MLOps". *International Journal of Artificial Intelligence, Data Science, and Machine Learning*, vol. 2, no. 2, June 2021, pp. 92-102
11. Benlarbi, Saida. "Estimating SLAs availability/reliability in multi-services IP networks." *International Service Availability Symposium*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006.

12. Muppaneni, Kavya. “Comparative Analysis of Client-Side Storage Mechanisms”. *International Journal of AI, BigData, Computational and Management Studies*, vol. 3, no. 1, Mar. 2022, pp. 171-82.
13. Gonzalez, Andres J., and Bjarne E. Helvik. "System management to comply with SLA availability guarantees in cloud computing." *CloudCom*. 2012.
14. Schmidt, Klaus. *High availability and disaster recovery: concepts, design, implementation*. Berlin, Heidelberg: Springer Berlin Heidelberg, 2006.
15. Muppaneni, Rajarshi Krishna. “How Enterprises Are Achieving 360° Customer Views With Dynamics 365”. *International Journal of AI, BigData, Computational and Management Studies*, vol. 2, no. 2, June 2021, pp. 129-38
16. Serrano, Damián, et al. "SLA guarantees for cloud services." *Future Generation Computer Systems* 54 (2016): 233-246.
17. Andrade, Ermeson, et al. "Availability modeling and analysis of a disaster-recovery-as-a-service solution." *Computing* 99.10 (2017): 929-954.
18. Gaddam, Rohit Reddy. “Hermetic ML Environments Using Conda-Lock and Docker”. *American International Journal of Computer Science and Technology*, vol. 3, no. 4, July 2021, pp. 22-34
19. Lu, Kuan, et al. "Fault-tolerant service level agreement lifecycle management in clouds using actor system." *Future Generation Computer Systems* 54 (2016): 247-259.
20. Clemente, Roberto, et al. "Risk management in availability SLA." *DRCN 2005). Proceedings. 5th International Workshop on Design of Reliable Communication Networks, 2005.*. IEEE, 2005.
21. Kumar Doodala, Appala Nooka, and Swathi Thatraju. “NLP-Driven Benefits Interpretation Engine for Personalized Member Communication”. *International Journal of Artificial Intelligence, Data Science, and Machine Learning*, vol. 3, no. 1, Mar. 2022, pp. 173-8
22. Garraghan, Peter, et al. "Emergent failures: Rethinking cloud reliability at scale." *IEEE Cloud Computing* 5.5 (2018): 12-21.
23. Parakala, Adityamallikarjunkumar. "Building Analytics-Driven Bots: RPA Meets Business Intelligence." *International Journal of Emerging Research in Engineering and Technology* 2.1 (2021): 77-87.
24. Lumpp, Th, et al. "From high availability and disaster recovery to business continuity solutions." *IBM Systems Journal* 47.4 (2008): 605-619.