Pre-Production Validation of Timing Failures in API–Event Financial Systems

Bhaskar Manchuri
bhasman@ieee.org

Independent Researcher
McKinney, USA

***Abstract*—Modern financial systems increasingly rely on synchronous REST APIs coordinating asynchronous, event-driven processing pipelines to meet scalability, reliability, and regulatory requirements. In these architectures, an API request may trigger downstream workflows involving message streaming, multi-service processing, and delayed state materialization before a response can be returned. While this pattern enables high throughput and service decoupling, it introduces significant challenges in meeting service-level objectives (SLOs), reasoning about failure semantics, and validating system behavior prior to production deployment. This paper presents a coordinated pre-production testing methodology for validating timing-sensitive failure modes in bidirectional API–event architectures used in latency-sensitive financial information systems. The study focuses on coordination failures that arise without explicit component outages, including late state propagation, timeout-induced false failures, and retry overlap with in-flight asynchronous processing. The methodology employs targeted fault injection and controlled integration-point disruption to deterministically reproduce these timing-related failure modes without modifying the application code or disrupting production-adjacent environments. Observations show that delays on the order of hundreds of milliseconds in asynchronous consumers or shared-state propagation can disproportionately degrade API tail latency and SLO compliance, even when downstream services remain healthy. The approach is compared against conventional integration testing, end-to-end testing, contract testing, and production chaos engineering to clarify its positioning within the broader validation landscape. Findings provide practical guidance for validating reliability expectations in event-mediated financial systems and offer a repeatable approach for pre-deployment assurance in regulated environments.**

***Index Terms*—Systems Integration, Event-Driven Architecture, Financial Information Systems, Fault Injection**

I. Introduction

Financial platforms increasingly combine synchronous REST APIs with asynchronous, event-driven processing pipelines to achieve scalability, operational flexibility, and service isolation [1], [2]. In these systems, a single API request may initiate a chain of asynchronous operations—message publication, downstream service processing, state materialization—before a synchronous response can be constructed and returned to the caller. This bidirectional API–event–API pattern has become prevalent in domains such as trade execution, payment processing, compliance reporting, and account management, where high throughput and auditability are simultaneously required [3].

While this architectural style provides significant benefits in terms of decoupling and horizontal scalability, it introduces a class of coordination challenges that are difficult to anticipate, reproduce, and validate using conventional testing approaches. Specifically, when a synchronous SLO depends on the timely completion of asynchronous pipeline stages, the system becomes vulnerable to timing-sensitive failure modes that do not manifest as explicit component outages. Instead, they emerge as coordination failures: late state propagation, timeout-induced false failures, and retry overlap with in-flight processing. Dean and Barroso [4] demonstrated that such tail-latency effects can be disproportionately amplified in multi-stage systems, even when individual components perform within expectations.

These failures are particularly consequential in regulated financial environments, where ambiguous outcomes can create reconciliation burdens, audit discrepancies, and downstream data integrity issues. Gunawi et al. [5] found that a significant proportion of cloud system failures arise not from individual component crashes but from complex interactions between otherwise healthy services. Industry-standard testing practices, including unit testing, contract testing [8], and end-to-end validation, are not designed to systematically exercise timing-dependent coordination boundaries. Production chaos engineering [6], [7], while effective at validating resilience against infrastructure-level faults, typically operates at a granularity that does not target the sub-second timing perturbations characteristic of these failure modes.

This paper presents a coordinated pre-production testing methodology that addresses this gap. The approach employs targeted fault injection at asynchronous integration boundaries to deterministically reproduce timing-related failures in controlled, non-production environments without requiring application code modifications.

Contributions:

- A taxonomy of timing-dependent failure modes in API–event financial architectures, distinguishing coordination failures from component-level faults.

- A coordinated pre-production testing methodology using calibrated fault injection for deterministic failure reproduction.
- A comparative analysis against conventional integration testing, end-to-end testing, contract testing, and production chaos engineering.
- Empirical observations on the relationship between asynchronous processing delays and API-level SLO degradation.

II. System Architecture

Figure 1 illustrates the reference API–event architecture under study.

A. Request Lifecycle

A typical request traverses: (1) API ingress under a defined SLO (e.g., p99 $\leq$ 500 ms) [10]; (2) event emission to a messaging system such as Apache Kafka [9]; (3) asynchronous consumer processing following patterns described by Hohpe and Woolf [1]; (4) state materialization to a shared data store; and (5) API response construction by polling the materialized state within the timeout window.

B. Coordination Dependency

The critical characteristic of this architecture is the *temporal coupling* between the synchronous response path and the asynchronous processing pipeline. Although the components are logically decoupled through messaging, the end-to-end request semantics require that asynchronous processing complete and state materialize within the synchronous timeout window. This creates an implicit coordination requirement that is not expressed in any individual component's interface contract and is therefore difficult to validate through conventional component-level or contract-based testing [8].

III. Timing Failure Modes

The timing-sensitive failure modes described below are distinguished from conventional component faults in that they arise from coordination timing rather than individual service unavailability—a pattern consistent with the distributed bug taxonomy of Gunawi et al. [5].

A. Late State Materialization

The asynchronous pipeline completes successfully, but the result is written to the shared data store after the API's polling window expires. Formally, let t_{emit} denote event emission time, t_{mat} materialization time, and t_{deadline} the API response deadline. A late materialization failure occurs when:

$$t_{\text{mat}} > t_{\text{deadline}} \quad \text{where} \quad t_{\text{deadline}} = t_{\text{emit}} + \Delta_{\text{timeout}} \tag{1}$$

B. Timeout-Induced False Failures

The API reports a failure to the client, but the asynchronous pipeline subsequently completes successfully. In regulated financial environments, such false failures create reconciliation discrepancies that may trigger audit findings under frameworks such as DORA [15] and OCC guidance [16].

C. Retry Overlap with In-Flight Processing

Client retries may overlap with the original asynchronous processing, producing ambiguous outcomes. Nygard [3] identifies this as a stability anti-pattern. Consequences depend on idempotency: fully idempotent pipelines waste resources; non-idempotent stages may produce duplicate transactions or conflicting state.

D. Cascading Timing Degradation

A modest delay in one consumer triggers backpressure in subsequent stages, amplifying total pipeline latency beyond the sum of individual delays. Dean and Barroso [4] characterize this as a tail-at-scale phenomenon; this work observes analogous amplification within event-mediated financial pipelines.

Figure 2 illustrates the delayed response and retry overlap scenario.

IV. Limits of Conventional Testing Approaches

A. Unit, Component, and Contract Testing

Unit and component tests verify local correctness but operate within a single service boundary and cannot express cross-service timing constraints. Contract testing frameworks such as Pact [8] validate structural interface compatibility but are inherently *structural* rather than *temporal*—they verify that a message conforms to a schema, not that it arrives within a timing window. The AsyncAPI specification [17] similarly addresses structural correctness of event interfaces rather than temporal coordination.

B. End-to-End Testing

End-to-end tests exercise the complete request path but are typically provisioned with deterministic, low-latency infrastructure that does not reflect production timing variability. They verify functional correctness rather than temporal boundary conditions.

C. Production Chaos Engineering

Chaos engineering [6], [11] validates resilience by injecting faults into production environments. However, it has key limitations for the timing failures addressed here: chaos experiments operate at infrastructure granularity (kill a process, drop a link) rather than sub-second timing precision; they are often constrained in regulated environments [15]; and they introduce stochastic perturbations rather than calibrated, deterministic delays [7]. Unlike chaos engineering experiments that introduce stochastic perturbations to evaluate recovery behavior, the proposed methodology focuses on deterministic timing perturbations designed to reproduce specific coordination failure modes.

Table I summarizes the coverage of each approach.

These testing approaches represent widely adopted industry validation practices in distributed microservice systems, particularly within CI/CD pipelines used in financial institutions.

The comparison highlights that existing validation approaches focus primarily on functional correctness, interface compatibility, or infrastructure resilience. None explicitly

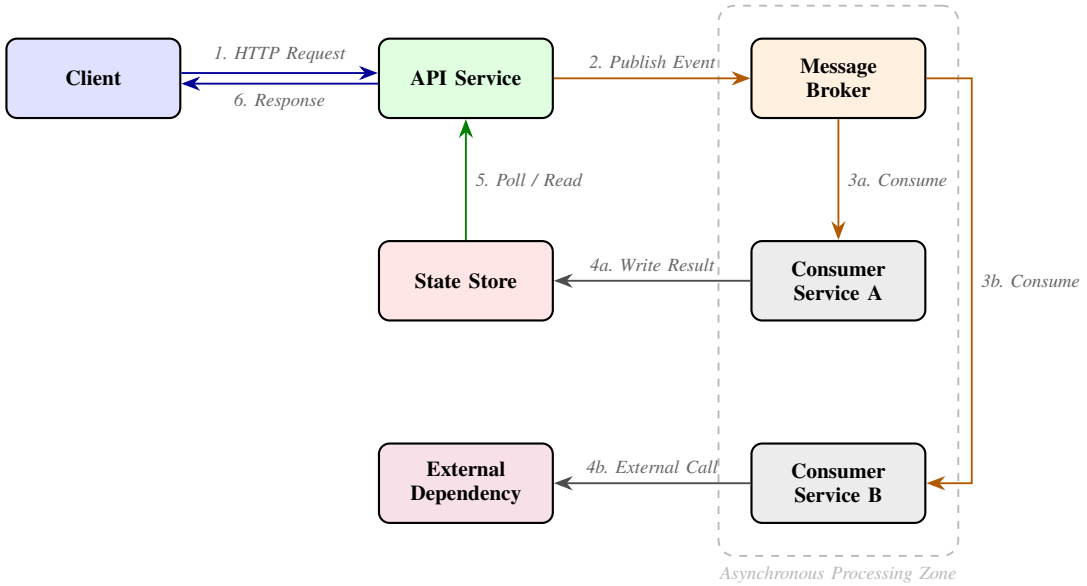

Fig. 1. Reference API–Event architecture. Numbered arrows trace the request lifecycle. The dashed boundary delineates the asynchronous processing zone.

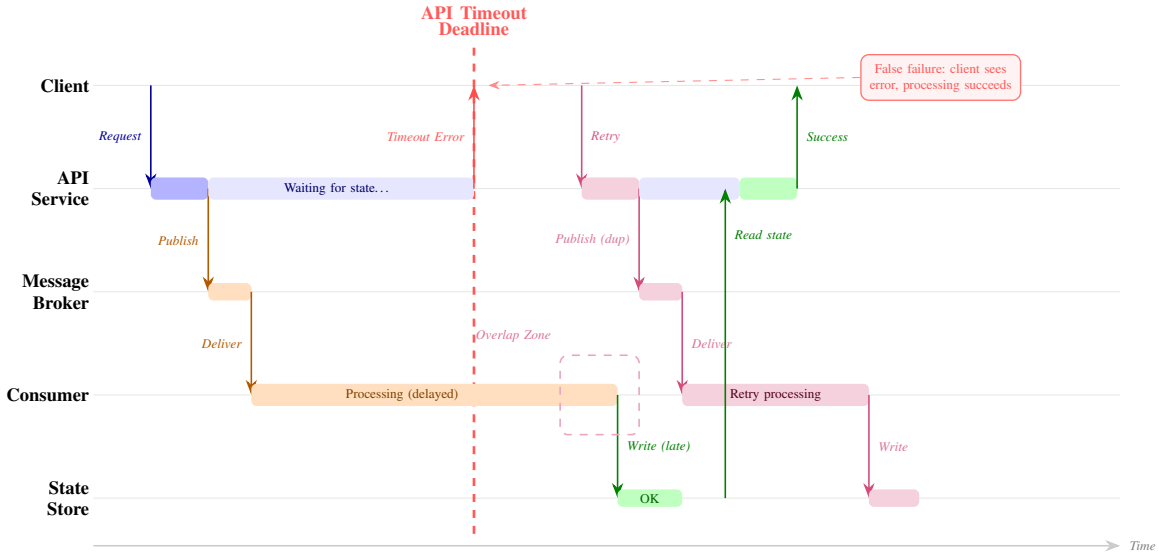

Fig. 2. Timeline of delayed response and retry overlap. Consumer processing exceeds the API timeout deadline (dashed red), producing a false failure. The client's retry overlaps with in-flight processing before the API reads the late-materialized state.

target timing-sensitive coordination boundaries where asynchronous processing must complete within synchronous deadlines. The proposed methodology complements these practices by introducing calibrated timing perturbations at integration boundaries, enabling systematic exploration of temporal failure conditions that remain invisible to conventional test suites.

V. Coordinated Pre-Production Testing Methodology

The goal of the methodology is to deterministically reproduce the coordination failure modes identified in Section III by introducing calibrated timing perturbations at asynchronous integration boundaries. The methodology is implemented in pre-production environments that mirror production topology and messaging infrastructure. Fault injection mechanisms are introduced as external infrastructure components (e.g., network proxies, message interceptors, or sidecar processes) and orchestrated through automated test scenarios executed within the release validation pipeline. This allows timing perturbations to be applied without modifying application source code or deployment artifacts. Tools such as Toxiproxy [18] and Chaos Mesh [19] provide primitives that support this approach.

A. Design Principles

The methodology is guided by four principles: (1) **Integration-point targeting**—faults are injected

TABLE I
COMPARISON OF VALIDATION APPROACHES AGAINST TIMING FAILURE MODES

Failure Mode	Unit/ Component	Contract Testing	End-to-End Testing	Chaos Engineering	Proposed Methodology
Late state materialization	No	No	Unlikely	Partial	Yes
Timeout-induced false failure	No	No	Unlikely	Partial	Yes
Retry overlap / idempotency	Partial	No	Possible	No	Yes
Cascading timing degradation	No	No	No	Partial	Yes
Component-level logic errors	Yes	Partial	Yes	No	No
Interface schema violations	No	Yes	Yes	No	No
Infrastructure availability	No	No	Partial	Yes	No

at boundaries between components rather than within component logic [3]; (2) **Calibrated perturbation**—delays are calibrated to specific magnitudes relative to the SLO budget [6]; (3) **Deterministic reproduction**—each scenario produces a known, repeatable outcome for a given delay configuration [7]; and (4) **Non-invasive execution**—no changes to application source code or deployment artifacts are required.

B. Fault Injection Techniques

Figure 3 illustrates the calibrated delay injection points (δ_1–δ_3) used to reproduce timing failures.

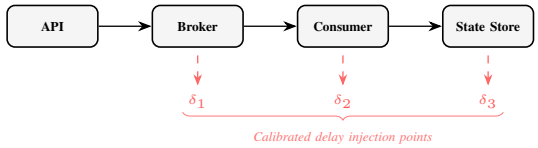

Fig. 3. Fault injection points (δ_1–δ_3) at message delivery, consumer processing, and state store write boundaries.

Four techniques are employed:

Message consumer delay injection. A proxy holds messages for a configurable duration before delivery, simulating slow consumer processing or broker delivery delays. The delay is applied uniformly or stochastically to simulate tail-latency behavior [4].

State store write latency injection. A middleware proxy injects write latency at the data store interface, simulating contention, replication lag, or cache invalidation delays [2]. This directly targets the late materialization failure mode.

Downstream dependency throttling. External dependency calls are intercepted and delayed, simulating degraded third-party performance without modifying those services. Toxiproxy [18] provides a representative implementation.

Selective API timeout reduction. The API's timeout budget is selectively reduced, narrowing the margin in Equation 1 and enabling sensitivity analysis at boundary conditions.

C. Execution Protocol

Each experiment follows a structured workflow. First, baseline measurements are collected without fault injection to establish reference latency distributions and SLO compliance. Next, a fault injection configuration is selected by specifying delay magnitude and injection point (δ_1–δ_3). The scenario is then executed under representative load conditions while the injection mechanism applies calibrated delays at the selected integration boundary. During execution, the system records the observability metrics described in Section V.E, including latency percentiles, SLO compliance rate, retry behavior, and outcome accuracy. After completion, results are compared against the baseline to identify threshold values at which specific timing failure modes emerge.

For compound scenarios, multiple injection points are activated simultaneously to evaluate nonlinear interaction effects across the asynchronous pipeline.

D. Test Scenario Design

Table II describes representative scenarios and their target failure modes.

TABLE II
FAULT INJECTION SCENARIOS

Scenario	Fault Injection	Target Failure Mode
Consumer delay	200–500 ms message hold	Late materialization, SLO violation
State store delay	100–300 ms write latency	False failure, retry overlap
Dependency throttle	300–800 ms call delay	Cascading degradation
Combined delay	Consumer + state store	Compound timing failure
Reduced timeout	API timeout reduced 30%	Boundary sensitivity

E. Measurement and Observability

Each test execution captures API response latency distributions (p50, p95, p99, max), SLO compliance rate following the error budget model [10], outcome accuracy (whether the API response matches the actual processing result), retry rates and outcomes, and state consistency after completion, consistent with distributed observability practices in modern microservice systems [20]. These measurements are correlated with injected fault parameters to establish sensitivity relationships and identify threshold values at which specific failure modes emerge.

VI. Observations

A. Delay Sensitivity

Consumer processing delays of 200 ms produced measurable increases in API p99 latency beyond the 500 ms SLO threshold. At approximately 400 ms of injected delay, SLO compliance dropped from near-baseline levels ($\sim$99%) to roughly 94–95%, with violations attributable to late state materialization.

State store write delays exhibited a sharper sensitivity curve. Because the API polls at fixed intervals, even modest write delays (100–150 ms) could cause a missed polling cycle, resulting in an effective delay equal to the polling interval—an amplification effect analogous to quantization behavior [21] not predicted from component-level analysis.

B. Outcome Accuracy Under Timing Stress

Under delays near the API timeout boundary, false failure rates reached approximately 15–20% when total processing time fell within 10% of the configured timeout threshold, suggesting a narrow ambiguity zone sensitive to minor timing fluctuations.

C. Retry Behavior and Idempotency

Retry overlap with in-flight processing was observed in a substantial fraction of cases. Fully idempotent pipelines produced no data integrity issues but consumed additional capacity. Partial idempotency gaps—particularly in external API call stages—produced detectable state inconsistencies requiring reconciliation.

D. Cascading Effects

Combined fault injection produced API latency degradation exceeding the sum of individual contributions. This superlinear behavior was attributed to thread pool saturation and polling amplification, consistent with tail-at-scale findings [4]. Timing sensitivity analyses must therefore consider compound scenarios rather than evaluating integration points in isolation.

E. Comparison with Production Incidents

The methodology successfully reproduced failure modes previously observed only in production, including the false failure pattern and polling-amplification behavior. Alvaro et al. [22] describe a similar philosophy of systematically surfacing fault paths otherwise discovered only in production.

These results demonstrate that coordination-sensitive failure modes emerge within narrow timing margins that are unlikely to be exercised by conventional test suites. System reliability therefore depends not only on component correctness but on validating the temporal behavior of cross-service coordination paths.

VII. Design Implications

A. SLO Budgeting Across Asynchronous Boundaries

SLO definitions must account for the full processing chain, including message delivery, consumer processing, state materialization, and polling latency [10]. Defining SLOs solely based on synchronous service performance creates a gap between stated guarantees and actual behavior.

B. Polling Interval and Timeout Co-Design

A system with a 500 ms timeout and 100 ms polling interval has at most five polling opportunities; a delay causing one missed cycle consumes 20% of the timeout budget. Notification-based architectures can reduce this amplification at the cost of additional complexity [2].

C. Idempotency as a Testable Property

Idempotency is commonly specified as a design requirement [1] but rarely validated under realistic retry conditions. This methodology treats idempotency as a *testable behavioral property*, extending the philosophy of Kingsbury's Jepsen project [23] to application-level idempotency in event-mediated systems.

D. Pre-Production Validation as a Compliance Artifact

DORA [15] and OCC guidance [16] increasingly require resilience testing of critical financial technology systems. The deterministic, repeatable nature of this methodology supports integration into formal change management and release validation processes.

VIII. Related Work

Chaos engineering [6], [7], [11] validates system resilience through production fault injection. Tools such as Gremlin [13] and LitmusChaos [14] provide infrastructure-level disruption primitives. This work is complementary: where chaos engineering validates recovery from faults, the proposed methodology validates *correctness of behavior* under timing variability in pre-production environments.

Research on distributed systems testing includes formal verification using TLA+ [25], [12], correctness validation via Jepsen [23], [24], and lineage-driven fault injection [22]. This work applies a similar philosophy—testing under adversarial timing conditions—to API–event coordination in financial systems, with emphasis on operational practicality.

Prior work on event-driven architecture validation has addressed schema evolution and consumer contracts [1], [17], focusing on structural correctness. SLO management research [10] has examined error budgeting and adaptive timeouts. This work contributes by demonstrating how SLO compliance can be validated under controlled timing perturbations before production deployment.

While these tools provide delay injection primitives, they do not prescribe coordinated scenario design, failure taxonomies, or systematic threshold identification for timing-sensitive coordination boundaries; the present methodology contributes these elements.

IX. Limitations and Future Work

The fault injection techniques operate at integration boundaries and do not model application-level contention (e.g., thread pool exhaustion, garbage collection pauses). The observations are based on a single reference architecture; broader validation across choreography-based patterns [1] and multi-region deployments would strengthen generalizability. Automated exploration of the timing parameter space using lineage-driven fault injection [22] and integration with continuous deployment pipelines as a release gate [26] are practical extensions for future work.

X. Conclusion

This paper presented a coordinated pre-production testing methodology for validating timing-sensitive failure modes in API–event financial systems. The methodology employs calibrated fault injection at asynchronous integration boundaries to deterministically reproduce coordination failures—including late state materialization, timeout-induced false failures, retry overlap, and cascading timing degradation—without modifying application code.

Observations demonstrate that sub-second delays can produce disproportionate degradation in API tail latency and SLO compliance, with nonlinear amplification under compound delay conditions. The methodology reproduced multiple classes of timing failures previously observed only in production. Comparison with conventional testing approaches establishes the methodology's complementary role in addressing temporal coordination properties invisible to existing test suites.

As financial systems continue to adopt event-driven architectures [2], systematic pre-production validation of timing-dependent behavior will become a critical component of reliability engineering practice.

References

[1] G. Hohpe and B. Woolf, *Enterprise Integration Patterns*. Addison-Wesley, 2003.

[2] M. Kleppmann, *Designing Data-Intensive Applications*. O'Reilly Media, 2017.

[3] M. T. Nygard, *Release It!*, 2nd ed. Pragmatic Bookshelf, 2018.

[4] J. Dean and L. A. Barroso, "The tail at scale," *Commun. ACM*, vol. 56, no. 2, pp. 74–80, 2013.

[5] H. S. Gunawi *et al.*, "What bugs live in the cloud?," in *Proc. ACM SoCC*, 2014, pp. 1–14.

[6] A. Basiri *et al.*, "Chaos engineering," *IEEE Softw.*, vol. 33, no. 3, pp. 35–41, 2016.

[7] C. Rosenthal and N. Jones, *Chaos Engineering: System Resiliency in Practice*. O'Reilly Media, 2020.

[8] Pact Foundation, "Pact: Contract testing for microservices," 2017. [Online]. Available: https://pact.io

[9] J. Kreps, N. Narkhede, and J. Rao, "Kafka: A distributed messaging system for log processing," in *Proc. NetDB Workshop*, 2011.

[10] B. Beyer *et al.*, *Site Reliability Engineering*. O'Reilly Media, 2016.

[11] Y. Izrailevsky and A. Tseitlin, "The Netflix Simian Army," Netflix Tech Blog, 2011.

[12] C. Newcombe *et al.*, "How Amazon Web Services uses formal methods," *Commun. ACM*, vol. 58, no. 4, pp. 66–73, 2015.

[13] Gremlin Inc., "Gremlin: Chaos engineering platform," 2023. [Online]. Available: https://www.gremlin.com

[14] CNCF, "LitmusChaos," 2022. [Online]. Available: https://litmuschaos.io

[15] European Parliament, "Regulation (EU) 2022/2554 (DORA)," *OJ EU*, 2022.

[16] OCC, "Third-party relationships: Risk management guidance," Bulletin 2013-29, updated 2020.

[17] AsyncAPI Initiative, "AsyncAPI specification v2.0.0," 2021. [Online]. Available: https://www.asyncapi.com/docs/reference/specification/v2.0.0

[18] Shopify, "Toxiproxy," 2023. [Online]. Available: https://github.com/Shopify/toxiproxy

[19] PingCAP, "Chaos Mesh," 2022. [Online]. Available: https://chaos-mesh.org

[20] C. Sridharan, *Distributed Systems Observability*. O'Reilly Media, 2018.

[21] J. L. Hellerstein *et al.*, *Feedback Control of Computing Systems*. Wiley, 2004.

[22] P. Alvaro, J. Rosen, and J. M. Hellerstein, "Lineage-driven fault injection," in *Proc. ACM SIGMOD*, 2015, pp. 331–346.

[23] K. Kingsbury, "Jepsen," 2014. [Online]. Available: https://jepsen.io

[24] K. Kingsbury and P. Alvaro, "Elle: Inferring isolation anomalies," *Proc. VLDB Endow.*, vol. 14, no. 3, pp. 268–280, 2020.

[25] L. Lamport, *Specifying Systems: The TLA+ Language*. Addison-Wesley, 2002.

[26] B. Burns *et al.*, "Borg, Omega, and Kubernetes," *ACM Queue*, vol. 14, no. 1, pp. 70–93, 2016.