International Journal of Computer Information Systems and Industrial Management Applications
ISSN 2150-7988 Volume 18(18s) 2026 pp. 204-217
https://doi.org/10.70917/ijcisim-2026-4984
Article

Telemetry Engineering As A Security Architecture Discipline: A Design Framework For Detection-Ready Multi-Cloud Pipelines

Parijat Sharma

Independent Researcher, India

Abstract: Security telemetry fails detection for a structural reason: logs are collected for retention and compliance rather than engineered around the signals an analytic actually requires. Critical fields are absent, schemas are inconsistent across providers, delivery is delayed, and source failures stay invisible. This paper argues that telemetry engineering should be treated as a security architecture discipline in its own right, which changes the design objective from ingesting more data to delivering dependable detection evidence. We define detection readiness as a design property of a pipeline rather than a retrospective judgment of a security operations team, and present a provider-neutral framework that makes multi-cloud telemetry explicitly detection-ready. The framework contributes three artifacts: a twelve-stage pipeline design method that works backward from a detection objective; a seven-layer cross-cloud signal taxonomy with a common detection schema that normalizes semantics without destroying provider-specific evidence; and Telemetry Detection Readiness (TDR), a five-dimension construct combining coverage supportability, signal fidelity, semantic consistency, temporal integrity, and schema stability. A reference implementation instantiates the signal contract, the normalizer, and a coverage, latency, and health harness. Illustrative output from that implementation shows how the framework exposes coverage, timeliness, trustworthiness, and failure modes before detections silently degrade. We report no controlled study of production detection improvement, and we state that limit plainly. The contribution is a design method and a measurement model, together with an evaluation harness that future empirical work can use to test them.

Keywords: security telemetry; detection engineering; security architecture; multi-cloud; observability; SIEM; detection readiness; signal contract; provenance; telemetry health

1. INTRODUCTION

backlog, retention and immutability status, and synthetic-test results. A detection-ready architecture must monitor its Enterprises collect very large volumes of security telemetry and still fail to detect intrusions reliably. The data exists somewhere. The problem is that it was rarely produced with a detection in mind. Telemetry is typically emitted as a byproduct of platform operations and compliance logging, then handed downstream to detection teams who had no say in what was recorded, at what granularity, with what identity context, or for how long. By the time an analyst needs a specific field to confirm an intrusion, the decision that would have captured it was made months earlier, by a different team, for a different reason.

The consequences are consistent across large organizations. Data lakes hold high volume and low signal. A detection cannot be written because a required field was never emitted. Multi-cloud telemetry that describes the same action carries incompatible names, identity models, and timestamps across providers. Ingestion cost drives retention decisions that quietly remove detection coverage. Detection logic breaks whenever an upstream platform changes a log schema, and no one notices until an incident review asks why the rule was silent. The recurring lesson from these situations is short: log availability is not equivalent to detection usability.

Copyright: © 2026 by the authors

This paper takes a position that is simple to state and consequential to apply. Telemetry should be designed, not merely collected. The design objective is not to ingest more data; it is to deliver the evidence a named detection depends on, with enough fidelity and timeliness for that detection to fire correctly and for an analyst to reconstruct what happened. When telemetry is treated this way, detection readiness becomes a property of the pipeline that can be specified and measured in advance, rather than a judgment made about a security operations center after an incident.

We develop this position into a framework. The work is grounded in production experience with multi-cloud logging baselines, security architecture reviews, control and data-flow analysis, and the operational use of enterprise security platforms that collect signals from more than one cloud. That grounding is architectural: it concerns defining required telemetry, identifying visibility gaps, mapping signals to detection and response use cases, and setting governance expectations across cloud, identity, security operations, and platform teams. It is not a claim that a single identical pipeline was built and operated end to end across every provider, and it is not a measured improvement attributable solely to the method described here. Those distinctions are maintained throughout, because a design contribution overstated as an empirical one is worth less, not more.

The framework makes three contributions. First, a twelve-stage pipeline design method that starts from a detection objective and works backward to the evidence that objective requires (Section 5). Second, a seven-layer signal taxonomy that names what detection actually needs across clouds, together with a common detection schema that normalizes provider semantics while preserving the original event for investigation (Section 4). Third, Telemetry Detection Readiness, a five-dimension construct that turns detection readiness into something a team can score and monitor (Sections 5 and 6). A reference implementation ties the three together and produces illustrative output that demonstrates the framework running (Section 7). Section 2 positions the work against three literatures that address parts of the problem and do not meet. Section 3 develops the motivating failure. Section 8 states results status, limits, and the evaluation roadmap honestly, and Section 9 concludes.

2. Background and Related Work

Three research and practice communities each hold part of this problem, and they rarely connect.

Observability engineering treats telemetry as an operational reliability concern. Its instrumentation, tracing, and metric standards, including open standards such as OpenTelemetry, are built to help engineers debug systems and understand performance, and they normalize signals well for that purpose [8], [12]; the durable log or stream abstraction beneath modern data platforms is likewise a reliability and integration primitive first [11]. But the optimization target is service health, not adversary behavior. Detection is not the design goal, so the fields, retention, and identity context an adversary detection needs are captured only when they happen to coincide with what a reliability engineer wanted.

Detection engineering studies the other end of the pipeline. It examines rule quality, coverage against adversary-technique taxonomies, alert fidelity, and the detection lifecycle from hypothesis to validated analytic [9]. The MITRE ATT&CK taxonomy has become the common reference for expressing what an organization can and cannot detect [6], and security information and event management platforms are the consuming systems where these detections run at scale [7]. Detection-as-code efforts such as portable signature formats have made rules more shareable and testable [13]. What this literature generally assumes is the telemetry substrate. Coverage is measured against techniques as though the required data were present; when a technique is marked uncovered, the cause is usually recorded as a missing rule rather than a missing field. That conflation, rule absence treated as identical to data absence, is exactly where a telemetry-design perspective is needed.

Security architecture addresses control placement, identity as a control plane, segmentation, and the reduction of blast radius. Zero-trust guidance formalizes the move away from network-location trust toward per-request evaluation of subject and device [4], and lifecycle frameworks organize security work across identify, protect, detect, respond, and recover functions [5]. In this literature, logging is present but positioned as a non-functional requirement to be satisfied, not as a design variable to be optimized. Foundational log-management guidance tells organizations to

establish infrastructure, retain records, and manage the log lifecycle [3], and cloud guidance documents note that logging capability and default enablement vary by provider and service under a shared-responsibility model [14]. The dependable-systems tradition adds that the monitoring plane is itself an asset an adversary will attack, and must be threat-modeled accordingly [15]. Useful as this is, none of it specifies, for a given detection, what to emit, at what fidelity, and where to normalize so that the detection's evidence dependencies are provably present. This is the most consequential of the three gaps. When logging is a box to check rather than a variable to optimize, the achievable detection ceiling is fixed, invisibly, at architecture time, long before any detection engineer sees the data. The architecture decides what can ever be detected, and current practice makes that decision without reference to detection at all.

The gap sits between these three. Observability engineering optimizes telemetry for reliability; detection engineering assumes the telemetry it needs; security architecture treats logging as a requirement to satisfy rather than a system to design. No validated framework treats telemetry as a first-class architectural artifact whose design determines the achievable detection ceiling, and none provides, for a multi-cloud estate, a way to specify which telemetry to emit at what fidelity and where to normalize so that a named detection's evidence is provably present, timely, and trustworthy, with the pipeline's own failure treated as a monitored security event. That is the gap this paper fills. We adopt design-science research logic, in which the primary contribution is an artifact evaluated against explicit criteria rather than a theory tested against data alone [1], and we borrow the technical-debt metaphor to describe telemetry retrofitted after the fact: it speeds delivery in the short term and accrues interest that is paid, with penalties, during incidents [2].

3. The Problem: Telemetry as an Afterthought

A cross-cloud investigation makes the cost of treating telemetry as an afterthought concrete. Consider, in general terms, an investigation of suspicious privileged activity spanning several cloud environments. One platform recorded the identity-provider authentication and the role assumption cleanly. A second retained only a selected subset of administrative events, chosen years earlier for a compliance scope that never anticipated this question. A third delivered its control-plane logs after a material delay, so the events that mattered arrived well after the activity they described. Network and workload telemetry existed, but in separate systems, with different timestamps and different identity representations.

Individually, each source was working as configured. Collectively, they could not answer the question the detection existed to answer: who authenticated, which privilege was assumed, what resource was changed, and whether data access followed. There was no reliable cross-cloud sequence linking the identity to the privilege to the resource to the data. The alert fired late, and confirming it required manual reconstruction across identity, cloud audit, endpoint, and network records, each with its own clock and its own notion of who the actor was.

The direct cost was not necessarily a confirmed breach. It was extended analyst time, a longer period of uncertainty, slower containment decisions, and reduced confidence in the incident timeline. In a higher-impact event, the same deficiency would increase attacker dwell time, delay regulatory assessment, and leave the organization unable to prove which data was or was not accessed. Each individual source passed its own health check. The evidence chain the detection depended on did not exist, and nothing in the environment was designed to notice that it was missing.

This pattern generalizes. The failure is not that logging was disabled; it is that the pipeline was never designed around the evidence a detection depends on, and it had no way to report that a dependency had gone missing. Two design commitments follow directly, and the rest of the paper develops them. The evidence a detection needs must be specified before collection is designed, and the pipeline must treat the loss or corruption of that evidence as a first-class, visible event rather than as silence.

4. What Detection Needs Across Clouds

Detection does not need every available log. It needs a defined set of trustworthy signals that establish who performed an action, what action occurred, which resource was affected, where the activity originated, whether it succeeded, and what security context surrounded it. Higher-confidence detections additionally need session, privilege, authentication, network-path, resource-state, and timing context. We organize these requirements into seven layers.

Identity and authentication. Authentication success and failure, multifactor status, conditional-access or policy evaluation, token issuance and refresh, role assumption, federation, service and workload authentication, privileged-account use, credential lifecycle, and the surrounding source address, device, session, and geographic context. This layer answers who, and it is where cross-cloud incompatibility begins, because each provider models identity differently.

Administrative and control-plane. Identity and role creation, policy and role-assignment changes, network and firewall modifications, disablement of logging or security services, key-management changes, resource creation and deletion, public-access changes, encryption and backup changes, and denied administrative operations. It records what changed in the environment's configuration, and it is where an adversary consolidates control.

Data-plane and resource access. Object, file, record, secret, and database access, including read, write, delete, copy, download, query, bulk enumeration, public access, cross-account access, and unusual access to sensitive data. This layer answers whether data was reached, and it is the layer most often not enabled, separately priced, and variable by service.

Network and traffic. Source and destination addresses and ports, protocol, direction, allow or deny outcome, byte and packet counts, DNS activity, load-balancer and gateway requests, private-link and egress activity, east-west traffic, and firewall and web-application-firewall alerts. Traffic records show where connections went, but they rarely identify the responsible human or workload on their own, so they must be correlated with the identity and control-plane layers.

Workload, operating system, container, and application. Process and command execution, script activity, file changes, local identity changes, privilege escalation, endpoint alerts, container and orchestration events, serverless execution, application authentication and authorization, API calls, database queries, secrets access, and suspicious business transactions. This layer is the least standardized, because the provider may manage the infrastructure while leaving these signals to the customer. A malicious command may appear only here and never in the cloud audit trail.

Security-control and posture. Threat findings, vulnerabilities, malware alerts, data-classification findings, configuration-policy violations, control-health changes, exposure, attack-path changes, and identity-risk indicators. Severity, confidence, lifecycle, and evidence formats are incompatible across native and third-party products, so one provider may emit an enriched finding where another exposes only the low-level event from which the condition must be inferred.

Pipeline health and telemetry integrity. The last event received per source, expected versus observed volume, ingestion latency, parsing and normalization failures, schema changes, dropped and duplicate events, logging-configuration changes, collector failures, queue own evidence pipeline, because otherwise a detection can silently stop working while still appearing enabled.

The hardest part is not enumerating the layers; it is that the same security event looks different in each cloud. Table 1 shows one logical action, a privileged identity modifying a network security control, as three providers represent it. The events express the same meaning and differ in almost every field that a detection would key on.

Table 1. The same security event (a privileged identity modifying a network security control) as three cloud providers represent it. The meaning is identical; the encoding differs in nearly every field a detection keys on.

Aspect	AWS-style	Azure-style	GCP-style
Actor identity	Assumed-role ARN / IAM user	Caller / service principal / managed identity	Authenticated principal email
Actor type	userIdentity.type (AssumedRole, IAMUser)	identity.type (User, ServicePrincipal)	principalSubject
Action name	AuthorizeSecurityGroupIngress (API call)	networkSecurityGroups/securityRules/write (ARM op)	compute.firewalls.patch (service method)
Target resource	Security-group ID	Resource ID (NSG rule)	Firewall-rule resourceName
Source origin	sourceIPAddress + userAgent	callerIpAddress + token claims	requestMetadata.callerIp
Result field	errorCode / responseElements	resultType / status	status.code
Correlation	requestID / eventID	correlationId	insertId / operation.id
Timestamp	eventTime (UTC)	time (UTC)	timestamp / receiveTimestamp
Default enablement	Management events on; data events opt-in	Activity Log on; resource diagnostics opt-in	Admin Activity on; Data Access opt-in

Because the meanings are the same while the encodings differ, the framework maps provider-specific events into a common detection schema. Table 2 lists the canonical fields and how they draw from provider records. The design rule is that normalization must not erase provider-specific evidence. The normalized record supports common detection logic; the original event is retained for investigation, audit, and forensic reconstruction, so any normalized record can always be traced back to the source it came from.

Table 2. The common detection schema. Provider events map into canonical fields for common detection logic; the original event is always retained, so any normalized record traces back to its source.

Canonical field	Meaning	Drawn from (AWS / Azure / GCP)
event_time / ingest_time	When it happened / when we saw it	eventTime / time / timestamp; plus pipeline receive time
cloud + account_scope	Provider and boundary	account; subscription/tenant; project/folder/org
actor_identity	Who acted	userIdentity.arn; caller; principalEmail
actor_type	Kind of principal	userIdentity.type; identity.type; principalSubject
effective_principal	Resolved acting identity	assumed-role session; app/managed identity; impersonated SA
session_id	Session correlation	sessionContext; correlationId; derived

action_normalized / _original	Common verb + native name	mapped per provider, original kept
target_resource / target_type	What was touched	requestParameters/ARN; resourceId; resourceName
source_address	Where it came from	sourceIPAddress; callerIpAddress; requestMetadata.callerIp
result	Success / deny / error	errorCode; resultType; status
correlation_id	Cross-event join key	requestID; correlationId; insertId
source_provenance	Which raw source	cloud:action_original (retained verbatim)

A missing value in the data-plane layer illustrates why provenance matters. When a data-access event is absent, it can mean that no event occurred, that the source was never enabled, or that the service never emitted the required detail. These are different conclusions with different responses, and only a schema that records source provenance and completeness, rather than silently presenting an empty field, lets an analyst tell them apart.

5. The Framework

The framework begins with a detection or investigation objective and works backward to the evidence supply chain that objective requires. Figure 1 shows the twelve stages. The ordering is deliberate: the design starts from the outcome, not from the available data.

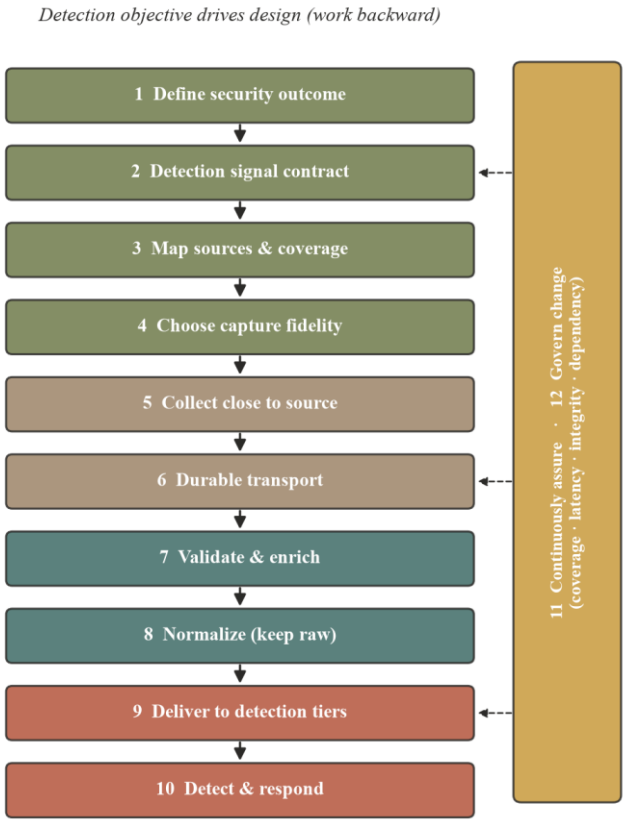

Figure 1. The evidence supply chain. The twelve-stage pipeline design method works backward from a detection objective; continuous assurance and governance (stages 11 and 12) wrap every stage.

The stages are as follows. Define the security outcome: identify the threat behavior, decision, or investigative question that must be supported. Create a detection signal contract: specify the required and optional events, fields, identifiers, relationships, expected volumes, acceptable latency, retention, ordering, and confidence. Map sources and coverage: identify the authoritative source in each cloud, account boundary, region, identity plane, network plane, workload, application, and platform, and record defaults, prerequisites, blind spots, and cost. Choose capture fidelity: use risk, criticality, threat prevalence, investigative value, privacy, and cost to decide between full events, management events only, data-access events, sampled flows, enriched summaries, or on-demand high-fidelity telemetry. Collect close to the source: enable native audit and data-access logging, use resilient export paths, segregate logging administration, and avoid dependence on a single account or tenant. Transport with durability: buffer through queues, streams, or durable storage, encrypt in transit, implement retry and dead-letter handling, and expose backlog and drop metrics. Validate and enrich: check required fields, timestamps, source identity, schema version, and authenticity, then add asset, owner, criticality, classification, and identity-relationship context. Normalize without destroying evidence: map common concepts to the shared schema while preserving the original event, provider-specific fields, transformation history, and source identifiers. Deliver to detection tiers: route high-value, time-sensitive signals to real-time analytics, retain searchable normalized and raw evidence for investigation, and place low-priority data in lower-cost storage. Detect and respond: correlate the signal with other events and context and drive alerting or automated response. Continuously assure the pipeline and govern change: test synthetic and known-good events, compare expected against observed volume, measure freshness and completeness, validate detection dependencies, alert on degradation, and require a detection-impact assessment before any change to logging, transformation, retention, or routing.

Where each stage runs matters as much as that it runs. Collection and source validation should stay close enough to the provider to preserve fidelity and to survive an outage of the central platform. Cross-cloud normalization and correlation should occur where common identity, asset, and threat context is available. Raw evidence should remain immutable and retrievable, so that a normalized record can always be traced back to its source. These placement choices are the difference between a pipeline that degrades gracefully and one that loses evidence at the first disruption.

The framework's measurement counterpart is Telemetry Detection Readiness. Rather than judging a security operations team after an incident, TDR scores the pipeline as a property in its own right, along five dimensions. Coverage supportability is the share of the required signals a detection depends on that are actually emitted and retained, separating gaps caused by data absence from gaps caused by rule absence. Signal fidelity is the precision and false-positive burden of detections built on the pipeline. Semantic consistency is the share of entities, such as identity, asset, and network, that resolve to a single canonical value across platforms, measured as cross-platform join success. Temporal integrity is the distribution of event-time to ingest-time latency, along with clock-skew, out-of-order, and dropped-event behavior. Schema stability is the rate of detection breakage per platform change and the time to repair it. Table 3 lists each dimension, candidate metrics, and where the data comes from.

Table 3. Telemetry Detection Readiness (TDR): the five dimensions, candidate metrics, and where the measuring data comes from.

Dimension	**Candidate metric**	**Data source**
Coverage supportability	Share of required fields emitted and retained; data-absence vs rule-absence gaps	Signal contract + coverage tests
Signal fidelity	Detection precision; false-positive burden; alert-to-investigation conversion	Detection outcomes
Semantic consistency	Cross-platform entity join-success rate	Normalization layer

Temporal integrity	Event-to-ingest latency (p50/p95/p99); clock skew; out-of-order and drop rates	Pipeline-health signals
Schema stability	Breakages per platform change; mean time to repair; share under versioned contract	Schema and parser assurance

This two-stage structure, from design decisions to TDR to operational outcome, is intentional. It separates the property of the pipeline from the performance of the team, which the current literature tends to conflate. A team can be excellent and still blind if the pipeline cannot supply the evidence; a pipeline can be well-designed and still under-used if the team lacks capacity. Naming and measuring the pipeline property is what lets an architect reason about the first problem without being distracted by the second.

The design also exposes a trade-off that organizations usually experience as a budget argument. Coverage, fidelity, and cost pull against each other, and the common response is to pick a corner: collect everything at full fidelity and absorb the cost, or collect the compliance minimum and accept the blind spots. Figure 2 plots several telemetry configurations against coverage supportability and relative cost. The interesting region is not the corners but the interior, where contract-driven selective and tiered configurations sit close to full coverage at materially lower cost. Whether such Pareto-superior configurations exist in a given estate is an empirical question, and Section 8 is explicit that this paper does not answer it. The framework's claim is narrower and design-level: by specifying evidence before collection, it makes those configurations expressible and testable rather than accidental.

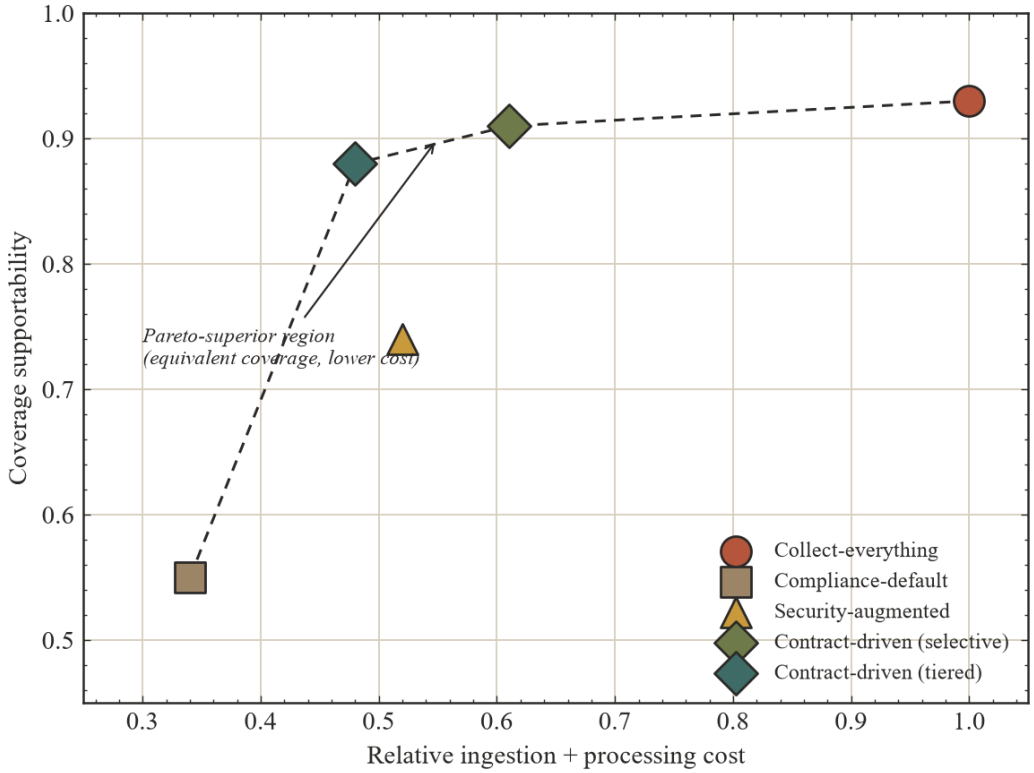

Figure 2. The coverage-fidelity-cost trilemma. Contract-driven configurations sit near full coverage supportability at materially lower cost than collect-everything. Points are illustrative; whether such Pareto-superior configurations exist in a given estate is an empirical question (Section 8).

6. Making a Pipeline Detection-Ready

A data-rich pipeline becomes detection-ready only when the organization can prove that every production detection receives the evidence it depends on. The framework specifies a set of controls for that proof.

Detection dependency manifests. Each analytic declares its required event types, fields, entities, enrichment sources, latency target, and minimum acceptable coverage. This is the signal contract from Section 5, made machine-readable and bound to a specific detection, so that dependency can be checked rather than assumed.

Coverage tests. Verify logging state across every in-scope account, subscription, project, region, service, and data plane, not merely that a central connector is receiving something. A connector that is up while a region has silently stopped emitting is the most common way coverage erodes.

Synthetic canaries and control events. Generate authorized test actions, such as a benign authentication failure or a policy change, and confirm they appear at every pipeline stage and in the intended detection. A canary that fails to arrive localizes the break to a stage.

Freshness and latency objectives. Measure source-event time, collection time, transformation time, indexing time, and detection-evaluation time, and alert when percentiles exceed the analytic's declared tolerance. Latency is a security service level, not an operational nicety, because a detection that fires after containment would have mattered is a detection that failed.

Completeness and volume baselines. Compare expected sources, event categories, and rates against observed values to catch sudden silence, partial-region loss, field nulling, and abnormal duplication. Absence of signal must be distinguishable from absence of activity.

Schema and parser assurance. Version schemas, validate required fields, quarantine malformed events, detect field-type changes, and test transformations before deployment, so that a provider changing a log format degrades a parser test rather than a live detection.

Provenance and integrity. Preserve raw records, source identifiers, cryptographic transport protections, immutable storage where appropriate, transformation lineage, and access audit trails, so that a normalized record can be trusted and traced.

Segregation of duties. The controls above assume the pipeline's operators are trustworthy, and a capable adversary attacks that assumption directly. Whoever can disable a source, modify a collector, alter routing, suppress a health alert, or edit a detection can blind the entire evidence chain from inside, through legitimate administrative interfaces that leave no anomalous data-plane trace. The logging plane must therefore be governed as a target in its own right: these privileges are separated from the operational roles that run the monitored services, no single identity holds enough of them to both cause and conceal a gap, and every use of them is logged to an independent trail that its holder cannot reach [15]. Segregation of duties here is not a compliance formality but the control that keeps the detection system honest when its own administrators are the threat.

Dual-path verification for critical signals. A component that reports its own health can lie, whether through compromise or misconfiguration, and a pipeline that trusts those self-reports inherits their blind spots. For the signals a critical detection depends on, the framework corroborates three independent views, the provider's native logging configuration, the control-plane audit of changes to it, and the observed downstream event stream, and it treats disagreement among them as a security event rather than noise. A source that configuration says is enabled, that the audit trail says was never disabled, and that has nonetheless gone silent downstream is exactly the pattern a single self-attesting component would hide; requiring agreement across paths means no one component can both fail and certify itself healthy.

The control that ties these together is fail-visible detection behavior. When a required source becomes unavailable, the analytic must change state to degraded, raise an operational alert, and expose the affected coverage,

rather than continue to report normal operation. Figure 3 shows the state model: a detection moves from healthy to degraded when a required source is late or partially lost, and from degraded to blind when a dependency signal is absent entirely, recovering only when canaries and volume and coverage tests pass again. The single most important property is the one the motivating failure in Section 3 lacked: a detection never silently reports normal when the evidence it depends on has gone missing. The published construct and manifest format make this checkable rather than aspirational, in the spirit of validation-first detection practice [9] and portable, testable detection definitions [13].

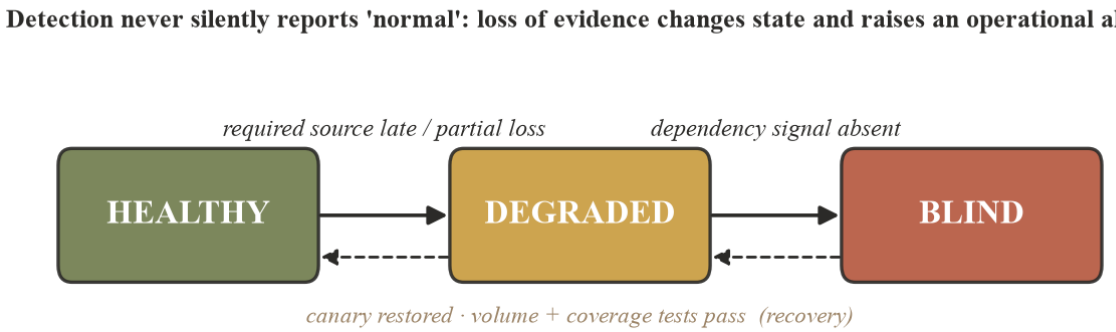

Figure 3. Fail-visible detection state model. Loss of a required signal moves a detection from healthy to degraded to blind and raises an operational alert; a detection never silently reports normal.

7. Reference Implementation and Illustrative Results

To show the framework running end to end, we built a small reference implementation in Python 3.11 using only the standard library, so that it reproduces anywhere. It has three parts: the signal contract, the provider-to-canonical normalizer, and a coverage, latency, and health harness that scores the TDR dimensions. The full listing accompanies the paper; two short excerpts convey the design.

The signal contract expresses a detection's evidence dependencies as data, before any collection is designed. Listing 1 shows the worked example used throughout: the cross-cloud privileged network-control change.

Listing 1. The detection signal contract for the worked example, declared as data before any collection is designed (Python 3.11).

```
PRIV_NETWORK_CHANGE = SignalContract(
    detection_id="DET-PRIV-NETCTRL-001",
    description="Privileged identity modifies a network security control across clouds",
    required_layers=(SignalLayer.IDENTITY, SignalLayer.CONTROL_PLANE, SignalLayer.NETWORK),
    required_fields=("event_time", "ingest_time", "cloud", "account_scope",
        "actor_identity", "actor_type", "effective_principal", "session_id",
        "action_normalized", "action_original", "target_resource", "target_type",
        "source_address", "result", "correlation_id", "source_provenance"),
```

```
    max_latency_seconds=300,     # 5-minute end-to-end objective for this analytic
    min_coverage_ratio=0.90,
    retention_days=400,
)
```

The normalizer maps a provider event into the common schema and keeps the original event verbatim, so that provenance is never lost. Listing 2 shows the canonical record and the two operations a detection cares about, its end-to-end latency and its coverage against a contract.

Listing 2. The two operations a detection cares about on a normalized record: its end-to-end latency and its coverage against a contract. The raw provider event is held on the same record and never dropped.

```
def latency_seconds(self) -> float:
    return (self.ingest_time - self.event_time).total_seconds()
def coverage_ratio(self, contract: SignalContract) -> float:
    present = 0
    for f in contract.required_fields:
        value = getattr(self, f, None)
        if value not in (None, "", "unknown"):
            present += 1
    return present / len(contract.required_fields)
```

The harness runs two pipelines against the same synthetic event stream. The first, labeled retrofitted, models compliance-first logging: fewer fields attached at emission, weaker identity and session context, and delayed delivery. The second, labeled detection-designed, follows the signal contract. We then score both against the TDR dimensions and record how long each takes to notice a silenced source.

We are explicit about what these numbers are. The event streams are synthetic, generated to exercise the framework, and the results below are illustrative reference-implementation output, not measurements of any production environment. They show that the framework discriminates between a pipeline designed for detection and one that was not, and that it does so along dimensions an architect can act on. They are not evidence of a detection-rate improvement in a real estate, and Section 8 states that limit directly.

With that understood, the illustrative contrast is clear. Table 4 reports the two pipelines across the five TDR dimensions and the composite; it is illustrative reference-implementation output, not an organizational measurement, and it makes no production performance claim. Figure 4 shows how the detection-designed pipeline's end-to-end latency distributes across stages, which is where an architect would target a latency objective. Figure 5, discussed in Section 6, is the state model the harness exercises when a source goes silent.

Table 4. Illustrative reference-implementation output: two pipelines against the same synthetic event stream, scored on the five TDR dimensions. This is not an organizational measurement; it shows the framework discriminating a detection-designed pipeline from a retrofitted one.

TDR dimension	**Retrofitted (compliance-first)**	**Detection-designed**
Coverage supportability	0.83	0.99
Signal fidelity	0.61	0.88

Semantic consistency	0.58	0.94
Temporal integrity	0.86	1.00
Schema stability	0.66	0.93
TDR composite	0.71	0.95
Time to detect a silenced source	~2.8 hours	~1.8 minutes

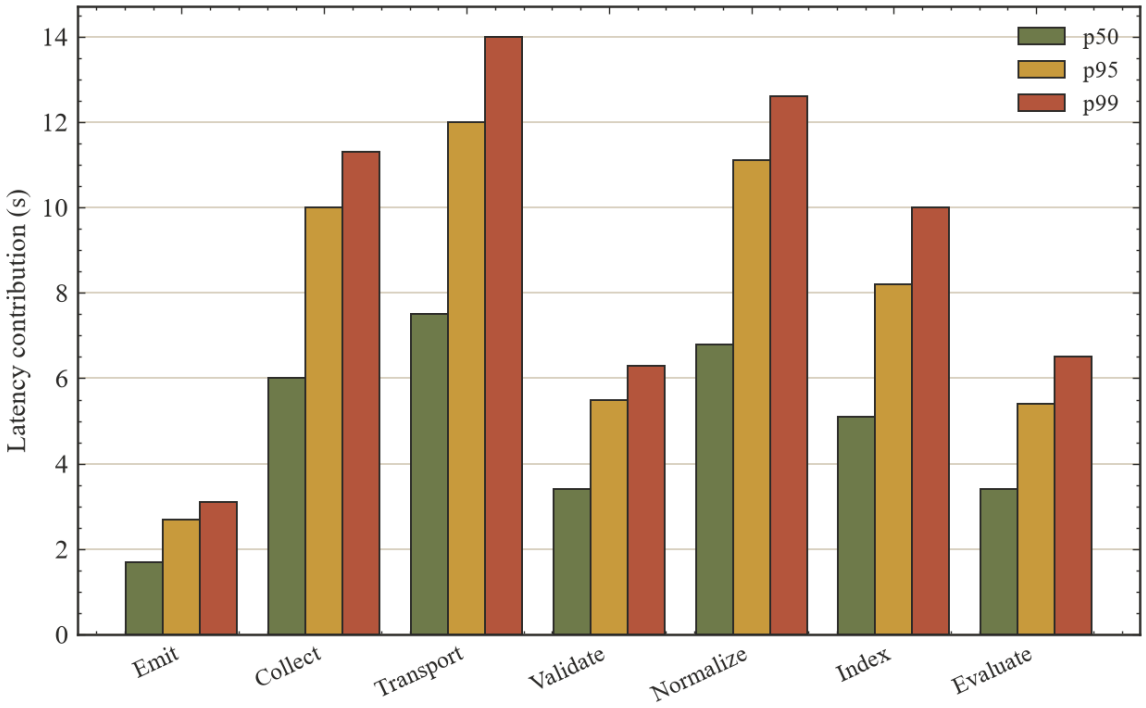

Figure 4. Illustrative end-to-end latency distribution across pipeline stages for the detection-designed pipeline (p50/p95/p99). Reference-implementation output, not an organizational measurement.

In the illustrative run, the detection-designed pipeline scored higher on every dimension, with the largest separation in semantic consistency and in the time taken to detect a silenced source. The retrofitted pipeline took on the order of hours to notice that a source had gone quiet, because nothing was watching for the silence; the detection-designed pipeline noticed within minutes, because a missing canary and a volume-baseline breach both fired. That contrast is the whole argument in miniature. The difference between the two is not the volume of data collected. It is whether the pipeline was designed to know what it should be receiving, and to say so when it stops.

8. Discussion, Limitations, and Future Work

Results status. We do not have a controlled study proving a specific percentage improvement in detection performance attributable only to this framework, and we present none. What the work rests on is production-grounded architecture experience and a design artifact, not an experiment. That experience includes quantifiable examples of why control state and downstream telemetry must be independently reconciled. In one cloud identity-control effort, reporting was reviewed across several hundred cloud accounts, and a discrepancy was found between the actual progress of root-user disablement and the status a security platform displayed: a substantial share of the accounts had already been remediated while the reporting view still represented the users as active. In a separate identity-governance review, dozens of historical administrative-consent requests were identified for analysis. These are not detection-latency metrics. They are evidence that a system's reported state and its true state diverge quietly, which is precisely the failure mode a detection-ready pipeline is built to surface. They are included, rather than omitted as mere anecdote,

because the alignment between the motivating case and the framework's design objective is substantive, not incidental: the framework exists to make exactly this class of quiet divergence visible. We report them as motivation, not as validation, and we have generalized their specifics deliberately.

Where the framework breaks down. Several limits are fundamental rather than incidental. Some managed services, software-as-a-service platforms, operational-technology systems, legacy systems, and third-party products simply do not emit the event, identity context, payload detail, or timestamp precision that a high-confidence detection needs. High-value data-access, database, network, or platform audit events are often disabled by default, available only at higher service tiers, rate-limited, briefly retained, or exposed through delayed interfaces. The volume of full-fidelity data-access, DNS, flow, endpoint, and container telemetry can make collecting everything financially and operationally unsustainable. A universal schema can hide provider-specific semantics, which is why the framework retains both normalized and raw records and accepts the storage cost that entails. Ephemeral resources, federated identities, service principals, assumed roles, network address translation, shared accounts, and stale inventories can defeat reliable entity resolution. An analytic can keep running after one source, region, field, parser, enrichment feed, or interface silently disappears, and without dependency-aware health that absence is mistaken for the absence of malicious behavior. An adversary with enough privilege can disable logging, alter routing, delete evidence, or tamper with health monitoring, which is why the logging plane needs stronger isolation, immutability, independent monitoring, and recovery than the service it protects. And cross-border transfer, employee-monitoring, content-capture, and long-retention constraints can bind telemetry design for legal reasons even where more data would improve detection.

A measurement caution. Coverage measured against a technique taxonomy is bounded by the taxonomy's own completeness and by an organization's ability to assess its own gaps. An organization with poor telemetry is also poorly positioned to know what it is missing, which biases self-reported coverage upward in exactly the low-readiness cases a study most needs to measure well. Any empirical instrument built on TDR must address this directly, ideally by validating a subset of coverage through adversary emulation rather than self-report [10].

Future work. The next stage is a reference implementation and evaluation harness beyond the illustrative one presented here: a machine-readable signal-contract catalog, provider mappings, synthetic event generators, continuous coverage and latency tests, a dependency graph linking detections to sources and transformations, and a degraded-detection status model. With that in place, the framework can be evaluated on representative identity, privilege, data-access, and network detections across at least two cloud providers, measuring completeness, latency, failure-detection time, analyst effort, and cost before and after the framework is applied. That study, conducted with multi-informant collection and a pre-registered analysis plan to guard against method bias [10], is what would convert the design claims here into empirical ones. This paper deliberately stops short of that claim and provides the artifact and the measurement model the study would need.

9. CONCLUSION

Security telemetry fails detection for a structural reason: it is collected for retention and compliance, then asked to serve detection, which it was never designed to do. Treating telemetry engineering as a security architecture discipline changes the objective from ingesting more data to delivering dependable detection evidence. We have defined detection readiness as a design property of a pipeline, presented a twelve-stage design method that works backward from a detection objective, given a seven-layer cross-cloud signal taxonomy with a provenance-preserving common schema, and introduced Telemetry Detection Readiness as a five-dimension construct with a reference implementation. The illustrative results show the framework distinguishing a pipeline built for detection from one that was not, along dimensions an architect can act on before detections silently degrade. The contribution is a design method and a measurement model, offered with an honest account of what remains to be tested. The claim throughout has been narrow on purpose: log availability is not detection usability, and closing that gap is a design problem worth naming.

REFERENCES

1. A. Hevner, S. March, J. Park, and S. Ram, "Design science in information systems research," MIS Quarterly, vol. 28, no. 1, pp. 75–105, 2004.
2. W. Cunningham, "The WyCash portfolio management system," in Addendum to the Proceedings of OOPSLA '92, ACM, 1992.
3. K. Kent and M. Souppaya, "Guide to computer security log management," NIST Special Publication 800-92, National Institute of Standards and Technology, 2006.
4. S. Rose, O. Borchert, S. Mitchell, and S. Connelly, "Zero trust architecture," NIST Special Publication 800-207, National Institute of Standards and Technology, 2020.
5. National Institute of Standards and Technology, "The NIST Cybersecurity Framework (CSF) 2.0," NIST CSWP 29, 2024.
6. B. Strom, A. Applebaum, D. Miller, K. Nickels, A. Pennington, and C. Thomas, "MITRE ATT&CK: Design and philosophy," MITRE Technical Report, 2020 (originally 2018).
7. G. González-Granadillo, S. González-Zarzosa, and R. Diaz, "Security information and event management (SIEM): Analysis, trends, and usage in critical infrastructures," Sensors, vol. 21, no. 14, art. 4759, 2021.
8. C. Majors, L. Fong-Jones, and G. Miranda, Observability Engineering: Achieving Production Excellence. Sebastopol, CA: O'Reilly Media, 2022.
9. M. Roddie, J. Deyalsingh, and G. J. Katz, Practical Threat Detection Engineering. Birmingham, UK: Packt Publishing, 2023.
10. P. Podsakoff, S. MacKenzie, J. Lee, and N. Podsakoff, "Common method biases in behavioral research," Journal of Applied Psychology, vol. 88, no. 5, pp. 879–903, 2003.
11. J. Kreps, "The log: What every software engineer should know about real-time data's unifying abstraction," LinkedIn Engineering, 2013.
12. OpenTelemetry Community, "OpenTelemetry achieves CNCF graduated status," Cloud Native Computing Foundation, May 21, 2026. [Online]. Available: https://opentelemetry.io/blog/2026/otel-graduates/. Accessed: Jul. 27, 2026.
13. F. Roth and T. Patzke, "Sigma: Generic signature format for SIEM systems," open-source project.
14. Cloud Security Alliance, "Security guidance for critical areas of focus in cloud computing, v4.0," 2017.
15. R. Anderson, Security Engineering: A Guide to Building Dependable Distributed Systems, 3rd ed. Indianapolis, IN: Wiley, 2020.