FRAMEWORK · COMPANION · V1.0

Observing and integrating the substrate

The API surface emerges from observability needs

Bearing south · Third piece · PDCA+ v2.0 · Public Review

What needs to be observable about a PDCA+ substrate in production — state, operations, flow, and the second-order property of the substrate observing its own observing — and how the API surface emerges as the answer to making those observations consumable by other systems and people.

AUTHOR

Joacim Brandell

Builds on Infrastructure Foundations and The Substrate, in detail · treats observability as the foundational concept and API as its consumable face · honest about the hard parts

Why observability before APIs

The previous pieces in the technical leg specified the substrate's physical infrastructure and its data model. With both in place, the question that follows is what the substrate looks like from outside — what other systems and people can see, what they can ask, what they can do, and how the framework supports being seen, asked, and acted upon. This is what API and integration discussions usually take up. This piece does too, but it takes them up in a particular order: observability first, API surface as a consequence.

The ordering matters. When the API surface is designed first, it tends to be shaped by the data model — endpoints corresponding to entities, operations corresponding to CRUD, queries shaped by what is convenient to expose. The resulting API is technically complete but often misaligned with what consumers actually need. When the API surface is designed from observability requirements, the shape is different: the API answers questions that someone genuinely needs answered, exposes operations that someone genuinely needs to perform, supports patterns of integration that are needed rather than possible. The same data model can produce two quite different API surfaces depending on which way the design pressure flows, and the observability-first direction produces better APIs.

There is also a structural reason rooted in the framework itself. PDCA+ requires observability as a non-negotiable property — the conjunction has to observe its own substrate, and the framework has to support second-order observation of that observing. The observability is the framework's contribution; the API surface is how that observability is made consumable. Starting with the API would obscure this dependency. Starting with the observability makes it visible.

This piece treats observability of the substrate, the API surface that emerges from it, the integration patterns that follow, and the hard parts that any honest treatment has to acknowledge. The reader who finishes this piece should be able to specify what their substrate needs to expose, what shapes of API would expose it well, and what kinds of integration their organisation needs to support. The final technical-leg piece — code-as-compliance — then has somewhere to run and something to observe; it depends on the foundations this piece establishes.

What needs to be observable

Observability of a PDCA+ substrate is not one concern. It is four, each addressing a different aspect of what the substrate is and does, each requiring different mechanisms, each producing different consumers and use cases. Naming them separately is what allows each to be addressed properly.

Substrate state

The substrate holds entities and their relationships. Substrate-state observability is the ability to query that holding — to ask what entities exist, what their current attributes are,

what their relationships to other entities are, and what their history has been. State observability is the foundational layer; everything else builds on it.

For PDCA+, substrate-state observability includes querying the chain (Regime Intent through Evidence Link), querying the supporting entities (Discipline Binding, Control, Taxonomy, Translation Rule, Measurement Construct), querying the identity-authority-accountability structure, and querying historical state at points in the past. The temporal dimension is consequential: "what did the substrate look like at the time this audit finding was generated?" is a common and important query, and answering it requires that historical state be retrievable rather than only inferable.

Substrate operations

The substrate does things. Substrate-operations observability is the ability to see what it does — what changes have been made, what classifications have been applied, what translations have been performed, what evaluations have produced what results, what routings have dispatched what to where. Each of these is a discrete operation; each leaves a trace; each trace is what makes the operation auditable, replayable, and analyzable.

The Substrate Change Record from the previous piece is the most prominent example, but operations observability is broader. It includes the classifications that a policy engine applies to incoming events, the translations that occur as content moves between regime vocabularies, the routing decisions that direct work to disciplines, the measurement evaluations that determine control effectiveness. Each is an operation the substrate performs on data flowing through it; each must produce an observable record.

Substrate flow

Events move through the substrate continuously. Substrate-flow observability is the ability to see this movement — to observe what events are arriving, where they are going, how long they are taking, where they are queuing, what is succeeding, what is failing. Flow observability is what makes the substrate's continuous operation legible at production scale.

Standard distributed-systems tooling for flow observability — distributed tracing, message-bus introspection, queue depth metrics — applies directly. What is specific to PDCA+ is the meaning of the flows: an event entering the event bus from event management discipline, classified through a policy engine, routed via the workflow engine to risk management, evaluated against measurement constructs, recorded in the substrate, and reflected back in the GRC tool is a specific kind of flow with specific structural meaning. Tracing that flow requires knowing what the events represent, not just that they exist.

Second-order observability

The fourth requirement is the one standard observability practice does not address. PDCA+ requires the framework to observe its own observing apparatus — to detect when classifications are drifting, when taxonomies are no longer fitting what they classify, when translation rules are producing inconsistencies, when authority is being invoked outside its

scope. This is second-order observability, and it builds on the first three but adds a structural commitment they do not have on their own.

The mechanisms are not exotic. Second-order observability requires that substrate state, operations, and flow be observable continuously rather than on demand; that patterns be computable across these observations rather than only individual events being inspectable; and that the framework have a way to act on what it observes about itself — by surfacing inadequacies to whoever has authority to address them, by triggering review processes when drift exceeds thresholds, by recording its own observations as part of the substrate so they can themselves be observed.

The four observability requirements together — state, operations, flow, and second-order — are what the API surface and integration patterns must serve. Each requirement produces specific demands on what the substrate exposes and how. The next sections walk through what those demands look like in practice.

> **THE FOUR OBSERVABILITY REQUIREMENTS**
> *State (what the substrate currently holds and historically held), operations (what the substrate does to data flowing through it), flow (the movement of events through the substrate's infrastructure), and second-order (the substrate observing its own observing apparatus). The first three are standard distributed-systems concerns; the fourth is what makes PDCA+ structurally distinct.*

The three pillars applied to the substrate

Standard observability practice rests on three pillars: logs (records of discrete events), metrics (aggregated quantities measured over time), and traces (causal paths through distributed processing). Each pillar maps to PDCA+ observability requirements differently, and each has substrate-specific content that distinguishes its application here from generic system observability.

Logs

Logs in a PDCA+ substrate are richer than typical application logs because the substrate's operations have substantive meaning. A log entry for "event classified as severity 3" is application-level; a log entry for "event classified as CMMC-relevant configuration deviation under taxonomy version 2.3, triggering routing to configuration-management discipline binding 47" carries substrate-specific meaning that makes it valuable to compliance reasoning, not just operational debugging.

Substrate logs need at minimum: the operation being recorded, the entities involved, the authority invoked, the time of the operation, the actor responsible, the substrate version against which the operation was performed. They need to be queryable by entity, by actor, by time range, by operation type, by authority — querying patterns much broader than typical application-log search. Modern log-aggregation tools (Elasticsearch with Kibana, Splunk,

Datadog Logs, Grafana Loki) can serve this if the substrate emits structured logs with substrate-specific fields; choosing the tool matters less than ensuring the emitted logs carry the structural information.

Metrics

Metrics for a PDCA+ substrate are partly the usual ones — request rates, error rates, latency distributions — and partly substrate-specific. Specific metrics that earn their place include: rate of substrate change records per entity type per day; rate of classifications producing each category; rate of translations between specific regime pairs; effectiveness measurements per control over time; rate of authority invocations per authority; rate of conjunction-surfaced inadequacies per category. Each of these tells a substrate-meaningful story that operational metrics alone do not.

Standard time-series databases (Prometheus, InfluxDB, TimescaleDB, AWS CloudWatch Metrics) serve well here. What matters is that the metrics expose substrate-specific dimensions rather than only operational ones. A metric that records "5,000 classifications today" is less useful than a metric that records "5,000 classifications today, broken down by taxonomy, category, source discipline, and disposition" — because the latter supports the kinds of questions second-order observation needs to ask.

Traces

Traces follow causality through the substrate. A single trace might begin with an event arriving from configuration management, follow it through schema validation, classification, routing, policy evaluation, workflow scheduling, human review (if required), substrate state update, and downstream notification to subscribers. Each step is a span; each span has its substrate-specific meaning; the complete trace tells the story of one finding's journey through the framework.

OpenTelemetry has emerged as the standard for distributed tracing instrumentation, and tools like Jaeger, Zipkin, and AWS X-Ray render the traces. For PDCA+, the spans need substrate-specific attributes — which entity, which classification, which authority, which regime — so that traces can be queried by substrate dimensions rather than only by service or endpoint. Tracing is what makes flow observability actionable; without it, flow observability is a list of events with no causal structure.

> **THE THREE PILLARS, SUBSTRATE-SPECIFIC**
>
> *Logs that carry substrate-meaningful operations, not just application events. Metrics that expose substrate-specific dimensions — taxonomies, categories, authorities — not just request counts. Traces that follow causality through the substrate with substrate-attributed spans. The tools are standard; what matters is what they are instrumented to capture.*

Second-order observability in practice

Standard observability tools support standard observability — they make state, operations, and flow visible. Second-order observability is something the framework itself implements on top of the first three, using the first three's outputs as its inputs. This section names what that implementation looks like in practice.

What second-order observability needs to detect

Several categories of drift are what second-order observability is structurally designed to surface.

Category drift in classifications. When the same kind of finding starts being classified into different categories over time without taxonomic justification, the taxonomy and the reality it classifies may be drifting apart. Detecting this requires aggregating classifications over time and looking for patterns — the same source discipline's outputs being classified differently this quarter than last quarter, the same kind of configuration deviation being assigned to different categories by different operators, classifications that are conceptually close to category boundaries occurring with increasing frequency.

Translation inconsistency. When translation rules between regime vocabularies produce inconsistent results — the same operational fact appearing differently when translated to one regime than to another, in ways that should not depend on the operational context — the translation rules may be incomplete or contradictory. Detecting this requires computing translations both forward and backward, comparing results, and flagging round-trips that do not converge.

Authority drift. When authority is being invoked in ways that diverge from the granted scope — decisions being made just inside the boundary, scope conditions being interpreted more broadly than the grant seems to support, delegated authorities being delegated further without explicit re-grant — the authority structure may be becoming nominal rather than operational. Detecting this requires correlating substrate change records with authority records and analysing scope-of-invocation against scope-of-grant.

Regime-binding integrity. When the chain from Regime Intent down to Evidence Link develops gaps — Requirement Records without satisfying Control Objective Records, Obligations without Evidence Links, Evidence Links pointing at artefacts that are no longer current — the substrate is starting to produce stakeholder entitlements that cannot actually be demonstrated. Detecting this requires periodic chain-walking with completeness checks.

None of these is detectable by looking at single events. All of them require pattern computation over many observations, and the patterns are substrate-specific rather than operational. The second-order observability implementation is what makes these patterns visible. In most implementations, it takes the form of a set of dedicated analyses running continuously over the substrate's logs, metrics, and traces — sometimes implemented as policy-engine rules, sometimes as dedicated detection jobs, sometimes as periodic batch reconciliations.

Surfacing what is detected

Detection is half of second-order observability; surfacing is the other half. When the framework observes an inadequacy in its own substrate, the observation has to reach someone who has authority to address it, in a form they can act on. This is itself a substrate operation — the conjunction's continuous-observation activity produces records that themselves enter the substrate, are addressable, and are governed by the same authority structure as everything else.

The surfacing mechanism typically takes the form of conjunction-finding records that point at the detected drift, summarise the pattern, indicate the entities involved, and identify the authority that should evaluate whether action is required. These records flow through the same workflow engine and event bus as everything else; the framework treats its own observations of itself as first-class operational events. This is what closes the second-order loop structurally: the framework's observations of itself are processed by the framework, not handled outside it.

> **SECOND-ORDER OBSERVABILITY, IN ONE SENTENCE**
>
> *The framework computes patterns over its own logs, metrics, and traces to detect category drift, translation inconsistency, authority drift, and regime-binding integrity failures — and surfaces what it detects through the same substrate that produced the observations, treating the conjunction's findings as first-class records in their own right.*

The API surface that emerges

With the observability requirements specified, the API surface follows. Each category of API exists because something needs to consume what the observability layer produces, or needs to contribute what the substrate consumes. The APIs are not exhaustive — every implementation will have product-specific endpoints — but the categories below are the ones the observability requirements demand. Each is shown as a category with representative operations; the actual surface in any implementation may have more operations within each category.

API · READ · Substrate state query

Operation	Shape	What it answers / does
GET entity by id	GET /entities/{id}	Retrieve current state of any substrate entity
GET entity at time	GET /entities/{id}?at=t	Historical state at point in time
Traverse chain	GET /chain/{intent_id}	Walk from any Regime Intent down through Evidence
Reverse traverse	GET /chain/from/{evid_id}	Walk from an Evidence Link up to the regime intents
Find by attribute	GET /entities/search?…	Locate entities by attributes or relationships
List versions	GET /entities/{id}/versions	All versions of a versioned reference entity

API · WRITE · Substrate change

Operation	Shape	What it answers / does
Create entity	POST /entities	Add new substrate content under invoked authority
Update entity	PATCH /entities/{id}	Modify with mandatory rationale and authority
Retire entity	POST /entities/{id}/retire	Mark as retired without deletion
Restore entity	POST /entities/{id}/restore	Reverse a retirement, with audit trail
Bulk operations	POST /changesets	Atomic multi-entity changes for blast-radius governance

API · STREAM · Event subscription

Operation	Shape	What it answers / does
Subscribe to entity events	`WS /events/entity/{type}`	Receive events as entities of a type change
Subscribe to discipline	`WS /events/discipline/{id}`	Receive events related to a specific binding
Subscribe to conjunction	`WS /events/conjunction`	Receive surfacing of inadequacies in real time
Replay from offset	`WS /events/replay?from=t`	Replay historical events from a point in time

API · COMPUTE · Evaluation

Operation	Shape	What it answers / does
Classify input	`POST /classify`	Apply policy-engine classification to an input
Translate between	`POST /translate`	Apply a translation rule to substrate content
Evaluate measurement	`POST /measure`	Compute a measurement-construct value
Check coherence	`POST /coherence`	Run coherence checks across affected entities

API · INSPECT · Audit and trace

Operation	Shape	What it answers / does
Query change log	`GET /changes?…`	Search the substrate change record by any dimension
Retrieve trace	`GET /traces/{trace_id}`	Full trace of one event through the substrate
Authority audit	`GET /audit/authority/{id}`	All invocations of a specific authority
Entity history	`GET /entities/{id}/history`	Complete change history of an entity

API · SURFACE · Conjunction findings

Operation	Shape	What it answers / does
List findings	`GET /findings?…`	Retrieve conjunction-surfaced inadequacies
Acknowledge finding	`POST /findings/{id}/ack`	Confirm receipt with the responsible authority
Resolve finding	`POST /findings/{id}/resolve`	Record the resolution with substrate changes made
Subscribe to findings	`WS /findings/stream`	Receive new findings as they are surfaced

The six API categories — state query, substrate change, event subscription, evaluation, audit and trace, and conjunction findings — together expose enough surface for the integrations the framework supports. Each category corresponds to one of the observability requirements: state and audit address state observability; event subscription addresses flow observability; change addresses operations observability; evaluation supports the substrate's policy-driven operations; conjunction findings expose second-order observations.

The shapes shown are illustrative. Real implementations will use GraphQL or gRPC or REST depending on context, may compose multiple categories into single endpoints for efficiency, may add product-specific operations the framework does not require. What matters is that each observability requirement has an API category that serves it, that the operations within each category are coherent, and that the surface taken as a whole is consumable by the kinds of integrations the next section describes.

> **THE API SURFACE, IN ONE SENTENCE**
>
> *Six categories of API operation — state query, substrate change, event subscription, evaluation, audit and trace, conjunction findings — each emerging from one of the four observability requirements. The specific transport and shape vary by implementation; the categories are what the framework's structural commitments require.*

Integration patterns

With the API surface specified, the patterns by which other systems integrate with the substrate become tractable. Three patterns are most common in PDCA+ implementations and worth describing in detail; many implementations will use all three concurrently.

GRC tool ↔ substrate

The GRC tool, as the infrastructure piece established, holds operational artefacts that the substrate references. The integration pattern is bidirectional. The GRC tool emits events when artefacts are created or modified — risk register entries, control assessment results, policy attestations, audit findings — and the substrate subscribes to these events through the event-subscription API to update relevant Evidence Link Records and Obligation Records. In the reverse direction, the substrate publishes conjunction findings, regime-binding changes, and control-objective updates that the GRC tool consumes to keep its operational view consistent with substrate state.

The pattern works best when the GRC tool treats the substrate as the system of record for everything chain-related (Regime Intent through Stakeholder Entitlement), while the GRC tool itself remains the system of record for operational artefacts. The risk register is the GRC tool's; the mapping from risk register entries to Control Objective Records is the substrate's. The audit finding is the GRC tool's; the connection between the finding and the regime requirement it implicates is the substrate's. Each system owns its primary records; integration is consumption of each other's events.

Operational discipline systems ↔ substrate

Configuration management systems, vulnerability management platforms, incident response platforms, change management tools — these are the operational discipline systems that produce most of the substrate's evidence inputs. The integration pattern is primarily one-directional: the operational systems emit events as their work proceeds, the substrate consumes the events, the substrate updates its state and records the operational reality. Less commonly, the substrate publishes contextual information back — current Control Objective expectations, regime classifications relevant to the discipline's scope, conjunction findings that affect the discipline.

For most operational disciplines, this integration takes the form of a streaming connection from the discipline system's event source to the substrate's event-subscription API, with classification and routing happening as events arrive. The substrate may apply transformation, classification under taxonomies, and translation between vocabularies as part of ingestion — turning operational events into substrate-meaningful records before they enter the chain. This is where the policy engine does much of its work, and where the boundary between operational reality and substrate representation is most consequential.

External systems ↔ substrate

External systems — auditor portals, regulator reporting systems, prime-contractor flow-down recipients, customer assurance requests — need access to evidence the substrate holds but typically in narrow, controlled, and audit-trailed ways. The integration pattern here uses primarily the audit, state-query, and chain-traversal APIs, often through a controlled gateway that adds authentication, authorisation, rate-limiting, and access-trail recording on top of the substrate's native APIs.

This is the integration pattern where the security and authority structure of the substrate matters most. External access must respect the authority records — an external auditor's read-only access to the chain is itself an authority that the substrate tracks, and the substrate's audit API records what external parties queried and when. For some implementations, the external integration is realised as a separate surface that filters and structures the substrate's responses appropriately, rather than exposing the native APIs directly.

> **THE THREE INTEGRATION PATTERNS**
>
> *GRC tool exchanges events with the substrate bidirectionally; operational discipline systems feed the substrate events one-directionally; external systems consume substrate state through controlled gateways with audit trails. Each pattern uses a subset of the API surface, and the patterns compose well because the API surface was designed coherently.*

The hard parts

No honest treatment of substrate observability and integration can omit the parts that are genuinely difficult. Each of the items below is a real engineering problem that implementations have to solve, and naming them is part of what makes this piece useful to the architects who will face them.

The hard part	What makes it hard
Cross-system identity	The same person or service exists in the substrate, in the GRC tool, in operational systems, and in enterprise identity. Keeping these linked correctly without coupling tightly to any single identity provider is genuinely hard. SCIM helps but does not solve. Most implementations end up with an identity-mapping layer that becomes load-bearing.
Eventual consistency	The substrate, GRC tool, and operational systems will sometimes disagree about state for windows of seconds to minutes. Architecting consumers to tolerate disagreement gracefully — and architects to communicate which view is authoritative for which question — takes real care. Many integration failures are eventual-consistency failures misdiagnosed as bugs.
Schema evolution at the API	The substrate's API surface will evolve. Versioning APIs while consumers continue running is a generic distributed-systems problem; doing it with substrate APIs that are consumed by long-running compliance integrations adds particular care. Most implementations use additive evolution (never break, only extend) and accept the cost of carrying obsolete fields.
Performance under load	Chain traversal is graph-query expensive; large taxonomies are costly to evaluate; substrate-change-record queries over years of history can be slow without careful indexing. Performance work tends to be needed exactly where compliance pressure is highest (audit cycles, regulatory submissions). Investing in performance early is cheaper than retrofitting it.
Secrets and credentials	API access requires credentials; credentials require lifecycle management; lifecycle management requires its own controls. The substrate's own access controls have to be a managed system, not an afterthought. Many production failures originate in expired credentials, leaked tokens, or overprivileged service accounts.
Multi-tenancy	Organisations with multiple business units, multiple subsidiaries, or multiple jurisdictions may need substrate isolation between them while preserving the conjunction's view across them. This is achievable but adds architectural complexity that should be designed in early. Retrofitting multi-tenancy is much harder than including it from the start.
Bootstrap and migration	Loading existing operational data into a new substrate is non-trivial — formats vary, classifications are inconsistent across systems, historical context is incomplete. The first substantial substrate population is often a multi-quarter effort, and underestimating it is a common implementation failure.

None of these makes implementation impossible; each is solvable with care. Naming them in advance is what allows them to be planned for rather than discovered at production-readiness review. An implementation team that has internalised this list early in scoping is better positioned than one that encounters each item as a surprise.

What this piece established, and the bridge to code-as-compliance

Observability of the substrate is four concerns: state, operations, flow, and the second-order observation that distinguishes PDCA+ from frameworks that observe only their first-order outputs. The three pillars of standard observability practice — logs, metrics, traces — serve the first three concerns when instrumented with substrate-specific meaning. Second-order observability is what the framework itself implements on top, computing patterns over the standard observability outputs to detect drift in classifications, translations, authority, and regime-binding integrity, and surfacing what it detects as first-class substrate records.

From these observability requirements, an API surface emerges in six categories — state query, substrate change, event subscription, evaluation, audit and trace, conjunction findings — each addressing a specific observability concern. The surface is consumable through three principal integration patterns: bidirectional with the GRC tool, one-directional from operational discipline systems, and controlled-gateway through to external systems. Each pattern uses a coherent subset of the API surface.

The hard parts remain hard. Cross-system identity, eventual consistency, API schema evolution, performance under load, secrets management, multi-tenancy, and bootstrap migration are real engineering problems that implementations have to plan for. Naming them in advance is part of what this piece offers; solving them is the work of implementation teams.

With observability and integration specified, the substrate has a complete production face. It is queryable, changeable, subscribable, observable in depth, integratable with other systems, and capable of observing its own observing. The final technical-leg piece — code-as-compliance — can now develop what runs against this surface. Code-as-compliance has somewhere to live (the policy engine), something to evaluate (substrate state), a way to be deployed (the change APIs with their authority structure), a path to integrate its findings (the conjunction-findings API), and an environment in which its operation is itself observable (the second-order layer). Each of these depended on what this piece established.

The technical leg, taken as a whole, then offers a complete operationalisation picture: infrastructure foundations as the physical layers, the substrate in detail as the data model and governance, observing and integrating the substrate as the production face, and code-as-compliance as the methodology for executable compliance rules. The four pieces compose into a buildable specification — not the buildable specification, because tool choices and organisational adaptations will vary, but enough specification that architects, GRC technology leads, and developers in this new era have a coherent foundation to work from.

WHAT THIS PIECE ESTABLISHED, IN ONE SENTENCE

Substrate observability has four requirements — state, operations, flow, second-order — addressed through standard practice (logs, metrics, traces) augmented with framework-specific second-order detection; the API surface that emerges has six categories; integration with the GRC tool, operational systems, and external parties uses three patterns over this surface; and the genuinely hard parts are nameable in advance so they can be planned for rather than discovered.