ifxtools

Document Version: 1.0 – 2026-08-13

Informix Diagnostic Agent: Assisted Monitoring for Large and Complex Estates

Airtool

Content

This document presents the **Informix Diagnostic Agent**: an automated monitoring and root-cause-diagnosis platform for large, complex IBM Informix estates: the multi-terabyte, high-concurrency, replicated systems (HDR / RSS / SDS clusters and Enterprise Replication meshes, on both Linux and AIX/POWER) where a single mis-tuned parameter or a latent engine defect can idle a national service. It is written for senior Informix specialists, and it makes one argument, stated up front in the manner of a strategic advisory: **the binding constraint on operating a large Informix estate is no longer knowledge of the engine. It is the systematic capture, at the moment of failure, of the evidence needed to reason about it.** The agent is built to remove exactly that constraint.

The platform reaches both the **engine** and the **host operating system** through a **single JDBC connection**: no SSH, no root, no host-resident daemon, no agent to install on the database server.

Over that one connection it collects **190 diagnostic probes** and a structured numeric snapshot, then runs them through a **diagnostic model**: a deterministic reasoning engine of per-probe signals, cross-metric rules and a causal-synthesis layer. Rather than merely reporting metrics, the model **reasons from symptom to root cause across layers** and returns a ranked, evidence-backed diagnosis that names the single thing to fix first.

It is, in effect, **the reasoning a senior Informix architect performs, captured as a model and run continuously**.

The same collection can also run **offline** from the host itself and be reconciled centrally, so an air-gapped or JDBC-restricted server is a first-class citizen. And because the model is exposed as an **MCP (Model Context Protocol) tool surface**, an LLM agent can drive it conversationally across a fleet: the encoded expertise is **callable by both people and machines**.

The headline. A read-only observer that never disturbs the patient. The agent:

- establishes **one** labelled, self-excluding connection;
- obtains `onstat` output through the SQL Admin API rather than a shell;
- reads host state as the unprivileged `informix` user;
- caps and time-bounds every read;
- and reasons only over **structured data it collected**, never over screen-scraped text.

On that foundation it categorises the estate across **18 engine-and-host diagnostic areas** and judges it with a diagnostic model of **98 per-probe signals and 58 rules** feeding a causal-synthesis layer. It **audits its own report for internal contradictions** before presenting it, and matches the running engine level against a **version-to-APAR knowledge base**.

And, uniquely, it **captures the full evidence set automatically at the instant a critical condition first appears**, closing the "we could not reproduce it" gap that dominates real-world Informix support.

The platform is delivered in **two complementary flavors**. The **diagnostic agent** described above `reasons`: it reaches in over one JDBC connection, periodically, and returns a ranked root-cause diagnosis. A second flavor, a passive **wire-tap**, `observes`: it runs on the database host and reconstructs every SQL statement and its timing off the network in real time, with no database login and no load on the engine — catching the still-running statement the trace ring cannot see until it completes. One tells you `what is wrong and why`; the other, `what is happening right now and what it costs`. They are one platform, and they interconnect: the wire-tap feeds its captures into the same reasoning pipeline. The second modality is detailed in its own section.

1. The Situation: Diagnosing an Estate You Cannot Pause

Monitoring an enterprise database estate has become structurally harder every year. A single production Informix engine now exposes on the order of **nineteen distinct diagnostic dimensions** — configuration correctness, CPU and virtual processors, shared memory, logs and checkpoints, disk performance and space, sessions and locks, per-user attribution, statement-level SQL, replication and backup, and the host operating system beneath all of it — each carrying hundreds of counters whose meaning depends on the others.

The estates that matter run **many such engines at once**, across coexisting versions and replication topologies, on both Linux and AIX/POWER, while the pool of engineers who can read all of that fluently is shrinking, not growing.

The result is a widening gap between the volume of signal a large estate emits continuously and the human attention available to interpret it — a gap no dashboard closes, because a dashboard shows numbers, not `diagnoses`.

This platform exists to close that gap: to collect the right structured evidence `safely`, reason from symptom to root cause across engine and host, and do so continuously and automatically, so the analysis is present at the moment it is needed rather than assembled by hand after the incident has passed.

Large Informix installations share a distinctive operational profile. They are:

- **old enough** to carry a decade of coexisting engine levels;
- **large enough** that a single instance holds hundreds of terabytes across thousands of chunks;
- **concurrent enough** that connection-pool storms and lock cascades are routine;
- and **critical enough** that the business cannot tolerate the instrument that diagnoses them adding load, taking locks, or, worst of all, being the reason the engine finally tips over.

They are also, almost universally, **replicated and heterogeneous**: HDR primary/secondary pairs extended with remote standalone secondaries (RSS) and shared-disk secondaries (SDS), Enterprise Replication meshes with Connection Managers arbitrating failover, running on AIX/POWER LPARs as often as on Linux.

Across such systems the classes of trouble are remarkably consistent, and they are the specification this platform is built to satisfy. They are also rarely located where they first appear: a CPU spike is usually a symptom of a bad plan, a connection-pool storm or a mis-sized configuration; a "disk is slow" complaint is often a plan reading far more than it should. Diagnosis therefore means **reasoning from a symptom to its cause across layers** (engine and host together), and that, in turn, requires having the right structured evidence at the right moment.

1.1. The complication: incidents are transient

A production database incident is, by its nature, a **transient event observed by people who arrive after it**. This single fact defeats manual diagnosis more reliably than any shortage of expertise:

- The **SQL-trace ring buffer is finite**; on a busy engine it can wrap in seconds, so the statements that caused an incident are overwritten before anyone connects to look.
- A **session avalanche** from an application connection pool resolves before a DBA can log in to observe it.
- A **checkpoint that blocks application threads** for minutes leaves only an aftermath in the message log, not the live state that would explain it.
- The standard remedy (arm `ifxcollect` / `AFCRASH` / `DUMPSHMEM` and "`collect diagnostics when it recurs`") depends on the recurrence being both timely and attended, which it rarely is.

The meta-problem

The consequence is a familiar pattern: incidents diagnosed by **inference rather than evidence**, and standing risks (a lock table quietly filling, an end-of-support engine, more virtual processors than the host can physically dispatch) that are **always visible in the data and never actually watched**. By the time a human looks, the evidence needed to reason about the failure has almost always evaporated. Removing that constraint, which means capturing the evidence automatically at the instant the condition appears, is the platform's founding purpose.

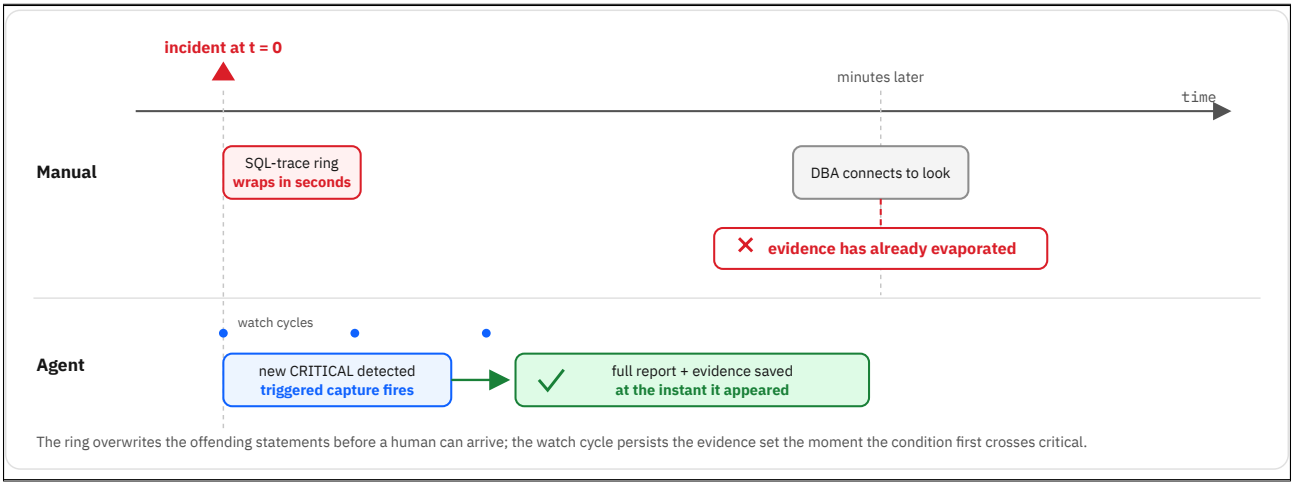

Figure 1 — The evidence-evaporation problem, and how triggered capture closes it

1.2. The failure taxonomy of a large Informix estate

The trouble in these systems resolves into a small number of recurring families. This taxonomy is the specification the platform's coverage is built to satisfy: every row maps to probes and rules enumerated later in this document.

Table 1 — Recurring failure families in large Informix systems

Family	Characteristic form
Engine defect / APAR	A known, already-fixed defect re-encountered on an un-patched level, often surfacing after a migration or fixpack change.
Backup / restore	Overwhelmingly control-plane failures (device, storage-manager, chunk preconditions), not engine faults.
ER / CDR replication	Queue backlogs, stale `syscdr`, apply threads blocked behind a user-defined routine.
HDR / RSS replication	Secondary drops, apply lag, a silent SMX stop, log-wrap resync failures.
CPU saturation	Almost always a `symptom`: a bad plan, a session storm, over-provisioned VPs, or a defect wearing a CPU costume.

Family	Characteristic form
Memory	Segment growth, out-of-memory crashes, pool corruption.
Locking	Lock-table storms, dictionary-mutex chains, long-held batch locks.
Bad SQL plan	Missing indexes, plan flips after statistics drift, optimizer-directive firefighting.
Configuration / tuning	VP classes, memory caches, physical-log buffer, lock table, stack size.
Disk space	Including sbspace `metadata` saturation, a trigger for smart-large-object corruption.
Checkpoint / physical log	Checkpoint-in-progress hangs; flush duration equal to the outage.
Session storm	Application connection-pool avalanches of thousands of sessions.
Network	Frequently a `consequence`, not a cause; the host NIC layer masquerading as an engine problem.
End-of-support exposure	Running an engine or OS level past its vendor support, a pervasive background risk.

> The recurring word in that table is **symptom**. "High CPU" can be a session avalanche, a missing index, a configuration error or a version defect: four different root causes wearing one costume. A tool that merely reports the symptom is of little use here; the requirement is to **reason from symptom to cause across layers**. That requirement drives the platform's synthesis layer, described later.

2. The Strategic Thesis: Why This Class of System Is Different

Positioned against the ways large Informix estates are monitored today, the agent occupies a deliberately different quadrant. The incumbent approaches are not wrong; they are structurally incapable of solving the evidence-evaporation problem, each for a specific reason.

Table 2 — Positioning against incumbent diagnostic approaches

Capability	Manual `onstat` / DBA scripts	Generic APM / infra exporters	On-demand IBM support + `ifxcollect`	Informix Diagnostic Agent
Engine **and** host in one view	Separate tools, separate logins	Host metrics; engine shallow	Engine-deep, host by request	**Both, one JDBC connection**
Root-cause reasoning (not just metrics)	In the DBA's head	Thresholds / dashboards	Human expert, after the fact	**Deterministic rules + causal synthesis**
Evidence **at the moment of failure**	Only if a human is watching	Sampled averages hide spikes	Almost never in time	**Triggered capture on first critical**
HDR / RSS / SDS + ER awareness	Manual `onstat -g`	None	Expert-dependent	**First-class probes + rules**
AIX/POWER parity with Linux	Hand-crafted per host	Weak on AIX	Yes, manual	**Per-platform probe specs**
Version-to-defect (APAR) matching	Tribal memory	None	The core IBM value	**Built-in APAR knowledge base**
Non-intrusive by construction	Depends on the DBA	Host agent installed	`ifxcollect` is heavy	**Read-only, labelled, self-excluding, time-bounded**
Machine-drivable (agentic)	No	Query API	No	**MCP tools + resources + prompts**

Three properties in that final column are the strategic core of the platform, and each deserves to be stated as a principle before the mechanics are shown.

Principle 1: Diagnose without interference.
The instrument must be safe to point at a system that is `already failing`. Every design decision in the collection layer subordinates convenience to the guarantee that the agent cannot be the marginal load that tips a struggling engine over. It uses one connection, labels and excludes its own traffic, reads through the SQL Admin API rather than a shell, and arms a socket deadline on every read so a runaway query is abandoned rather than left to burn CPU.

Principle 2: Reason only over plain, structured data.
The rules never parse `onstat` screen text. Collection produces a structured numeric snapshot and canonical tab-separated evidence; the reasoning layer consumes that. The identical snapshot can be produced **live over JDBC** or **offline by a shell collector on the host**, and the two are proven byte-for-byte equivalent. Plain data is portable data: it survives being e-mailed, stored, re-analysed months later, and reasoned about by a machine.

Principle 3: Make the capability agentic.

The diagnosis is exposed over MCP so that an LLM can list servers, run a probe, explain a query plan, and request a full triage, conversationally, across a fleet, gated by OAuth. The expertise encoded in the rules becomes callable, not just readable.

3. Architecture at a Glance

The platform is a layered Java 21 application (package root `com.informix.agent`). Each layer has a single responsibility and depends only on those beneath it; the Informix-specific behaviour (`onstat`, `sysmaster`, the SQL Admin API) is isolated in the connection and admin layers beneath a shared host, analysis, report and console stack.

Table 3 — Architectural layers, bottom to top

Layer	Principal types	Responsibility
Connection	`util.JDBCConnectionWrapper`	Drop-in `Connection` with execute/split/dialect helpers, statement self-tagging and a per-read socket deadline.
Admin API	`InformixAdmin`	`onstat()` / `task()` as text; `admin()` action commands resolved via `command_history`.
Host OS	`os.SystemAgent` / `InformixSystemAgent`	Run host commands as the unprivileged `informix` user over the same connection.
Monitoring	`probe.*` (`Category`, `Probe`, `Diagnostics`)	The probe catalog and its runner; per-probe failure isolation.
Analysis	`analyze.*` (`MetricsSnapshot`, `RulesEngine`, `Synthesis`, `Triage`)	Structured metrics to findings to a ranked, narrated diagnosis.
Sessions	`session.SessionStore`	Persist each triage (raw data + analysis + report); list, reload, re-analyse offline.
Configuration	`config.*`	Encrypted multi-server registry (AES-256-GCM), key provider, connection manager.
Reporting	`report.TriageDocXml` / `ReportRenderer`	One XML document to self-contained HTML and to PDF via Apache FOP.
Web console	`web.*` (Javalin + Carbon)	Server-rendered admin console; login-gated, offline-capable CSS.
Agentic	`mcp.*` + `oauth.*`	MCP JSON-RPC tool surface gated by an OAuth client-credentials grant.
Service	`watch.*` + `ingest.*`	Periodic triage, alerting, and transport-agnostic bundle ingestion.
Packaging	`cli.*`	picocli CLI, shaded fat-jar, Docker image.

3.1. The single-connection reach

The defining architectural fact is that **every layer above reaches both the engine and the host through one JDBC connection**. There is no second channel to secure, no host agent to deploy or patch, no credential distributed to the operating system.

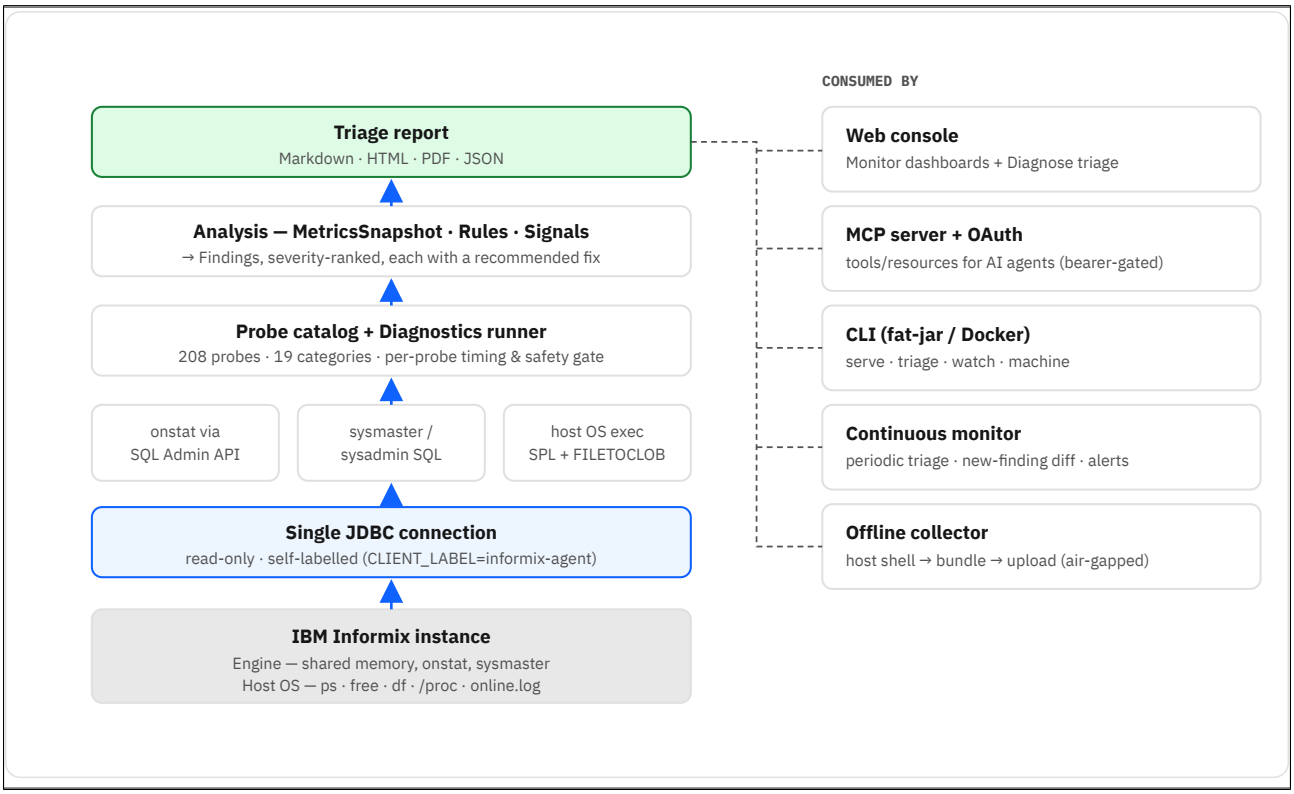

Figure 2 — Layered architecture: one JDBC connection feeds the probe, analysis and report layers, consumed by five operator surfaces

4. Non-Interference by Construction: the Collection Model

This section is the technical heart of the platform, because non-interference is the property that earns the right to run against a production system in distress. Each mechanism below is a deliberate concession of convenience to safety.

4.1. onstat without a shell: the SQL Admin API

The agent obtains `onstat` output as the return value of an SQL function, not by opening a shell on the host. The canonical form materialises the report text and strips its fixed-width padding:

```
SELECT TRIM(sysadmin:task('onstat','-g cluster')::char(32000)) AS cmd
FROM   sysmaster:sysdual;
```

Informational admin functions return their report `as the function's value`; the `::char(32000)` cast forces materialisation and `TRIM` drops the padding. Action commands go through `admin()`, which returns a numeric `cmd_number` (negative on failure) resolved to a human message via `sysadmin:command_history`. Every call is `sysadmin:`-qualified so it resolves cross-database while the session remains attached to `sysmaster`. The structured facts (cache ratios, wait profiles, chunk service times, dbspace fullness) come instead from `sysprofile`, `sysshmvals`, the dbspace/chunk views and the SQL-trace views, as numbers.

4.2. Host state as the unprivileged informix user

Where host facts genuinely need a shell (`ps`, `vmstat`, `svmon`, `/proc`, the `online.log` tail), the agent installs a single small SPL routine, `sys_os_exec`, and drives it over the same connection. The routine runs the command through the SPL `SYSTEM` statement, captures combined stdout/stderr to a per-session temp file, reads it back with `FILETOCLOB`, and deletes it immediately (and on exception).

> ### On-server footprint and posture
>
> The footprint is **one UDR** (`sys_os_exec(LVARCHAR(4096)) RETURNING CLOB`) plus a temp file that exists only for the duration of a single command. The routine runs as the `informix` user, **never root**, and the agent uses it only for read-only inspection. `LC_ALL=C` is pinned inside the routine so a localized host (a Spanish-locale AIX box answering `"promedio de carga: 0,49"`) does not emit comma-decimals or translated headers that would defeat the parsers. `uninstall()` drops the routine. The equivalent capability exposed to operators (the MCP `os_exec` tool) is **off by default and not even advertised** unless explicitly enabled.

4.3. The agent never reports itself: self-labelling

A monitoring session that appears in its own results as "the heaviest user" is worse than useless. The agent excludes its own traffic at two independent levels.

- **Session level.** The JDBC URL sets the connection property `CLIENT_LABEL=informix-agent`. Attribution queries exclude any session whose `sysmaster:sysenvses` carries that label, so the agent's own connection is never counted as workload.
- **Statement level.** The wrapper injects the marker `/* informix-agent */` into every SELECT/EXECUTE, deliberately `after` the leading verb, because the engine strips a comment that `leads` a statement from the stored trace text but preserves an inner one (verified live). Every trace-reading surface then filters

`sql_statement NOT LIKE '%/* informix-agent %'`, so the agent's statements never pollute the "top SQL" rankings even after the session has disconnected.

4.4. A runaway read must never become the problem

Query timeouts on the Informix JDBC driver were characterised empirically against a live 15.x engine. The conclusions are counter-intuitive and are encoded directly into the wrapper:

Table 4 — Empirical query-timeout behaviour (driver 4.50.10 vs live 15.x)

Mechanism	Result
`Statement.setQueryTimeout()`	Accepted but **never fires**: the statement ran at full engine CPU until externally killed.
`Statement.cancel()` from another thread	Returns cleanly, **interrupts nothing**.
`Connection.setNetworkTimeout()` (socket deadline)	**Works**. It aborts the blocked read near the bound. But the connection is then poisoned: any further call on it can hang forever.

The wrapper therefore arms a socket deadline per execute via a dedicated daemon executor, and on a trip it **sacrifices the connection**: it never touches it again (no close, which would itself hang), abandoning it to garbage collection and reopening. The metrics sweep and each probe carry their own scoped deadline. The honest cost is one discarded connection; the benefit is that a monitoring query can **never** be the reason an engine stays pinned.

4.5. Bounded capture, disabled-by-default tracing

Statement text is captured with an explicit cap (`SUBSTR(sql_statement,1,N)`, default 2,000 characters, clamped to the range 200-32,000), so a pathological query text cannot bloat a snapshot (a real bundle once carried a 321 MB probe dump before this bound). The **SQL-trace ring buffer is OFF by default**; the statement-level probes report empty grids until it is explicitly enabled, and the agent `measures the trace window itself` (`traceWindowSecs`, ring-wrapped detection) so it can warn when a ring holding only seconds of history voids any trace-based ranking. Observability is never assumed; it is checked and reported.

When tracing `is` enabled for a run, the ring is **sized against live memory, not guessed**. The trace ring costs `ntraces x per-statement-buffer` of shared memory, charged against the same `SHMTOTAL` cap the engine runs under. A single sizing model (`TraceProfile`) reads the live headroom to that cap, targets a useful history window at the observed statement rate, holds the ring inside a conservative fraction of headroom, and **hard-refuses** a ring that would cross the budget. When memory limits cannot be read it falls back to a fixed absolute ceiling rather than allocating blindly. The instrument an operator reaches for to `diagnose` a struggling engine can therefore never be the ring that tips `SHMTOTAL` over.

Fidelity of the capture is now treated with the same rigour as its size. Three mechanisms, added since the first release, make a trace-based ranking mean what it says:

- **An isolated trace phase.** Tracing is held `off` for the entire metrics collection and probe sweep, and switched on only for a single dedicated capture window at the end of the run. The rest of the triage therefore never competes with, or pollutes, the ring. The agent rewrites the `ix.stmt.trace_status` grid so it reports the capture window that was actually taken, not the "OFF" state that held while the probes ran.
- **A pre-eviction snapshot.** On a busy engine the ring overwrites its own oldest entries continuously, so a statement's row can be evicted between the moment its cost is ranked and the moment its text or plan is read, the classic "the plan shown under the wrong statement" corruption. The agent snapshots the live ring into a frozen temp **before** the engine evicts, at a conservative **50-70% of** `ntraces`, and reads every ranking, fingerprint and plan from that frozen copy. A ring that wraps mid-capture raises an explicit **corruption alert** rather than yielding a plausible-but-wrong ranking.

- **Worst-of-N capture.** An interactive run can capture over **several successive windows** and keep the worst observation per statement, so an expensive statement that fired in only one window is not averaged away by the windows in which it was idle, the same "worst observed moment" philosophy applied to statement forensics.

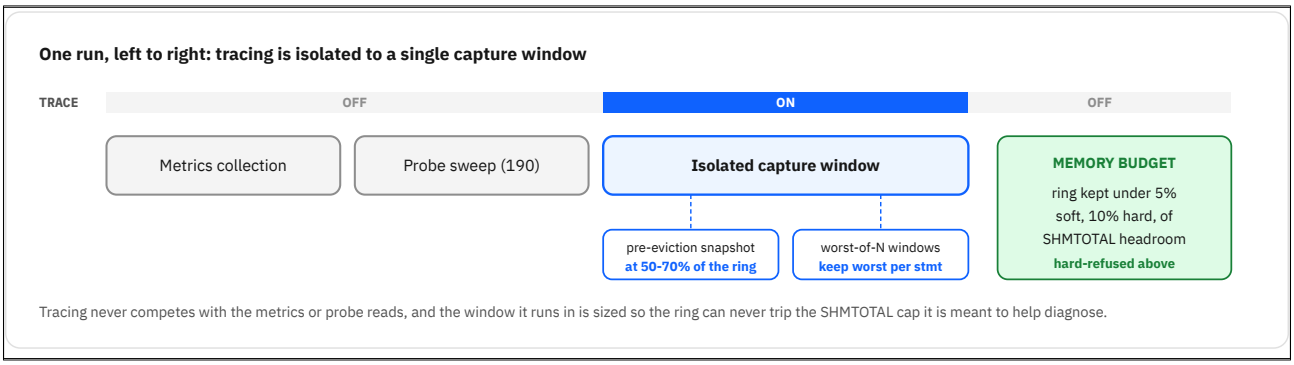

Figure 3 — The isolated SQL-trace phase: tracing is on only for a single, memory-budgeted capture window

Preventive memory-safety gate

The same principle is enforced before collection even begins. When allocated shared memory is already close to the `SHMTOTAL` ceiling, the live triage **halts with a single CRITICAL "memory nearly exhausted" finding** instead of running the heavier probes, and the offline collector's shell **preflight refuses to proceed** for the same reason. An engine that is one allocation away from failing is exactly the engine a naive monitor would push over the edge; here the instrument stands down and says why.

4.6. Read-only by construction, and a bounded footprint

The properties above add up to a footprint a change-review can approve on sight. The agent's session runs at **DIRTY READ** isolation, so it never takes, and never waits behind, a lock, the one behaviour that would let a monitor deadlock the workload it is inspecting. During a triage it issues only **informational** admin functions (the `onstat` family) and plain `sysmaster` / `sysadmin` SELECTs; the single state-changing operation anywhere in the platform is the **operator-gated SQL-trace enable**, which is disclosed in the run parameters and memory-budgeted as described above. Host commands are read-only inspections run as the unprivileged `informix` user, never root.

Table 5 — The collection footprint, in concrete terms

Bound	Value
Connections held	One, labelled and self-excluding; abandoned and reopened on a deadline trip, never left half-dead.
Isolation level	**DIRTY READ**, so no lock is taken or waited on.
Write operations	None during triage, except the operator-gated, disclosed SQL-trace enable.
Captured statement text	Capped at 2,000 characters (configurable, hard ceiling 32,000).
SQL-trace ring memory	Kept under 5% of `SHMTOTAL` headroom (soft), hard-refused above 10%.

Bound	Value
Per-read deadline	Graceful abort no sooner than 30 s, a socket backstop 10 s later.
On-server objects	One optional UDR plus per-command temp files removed immediately; dropped on `uninstall()`.

5. Diagnostic Coverage: the Category Model and the Probe Catalog

Coverage is organised as a two-level taxonomy. At the top are two **domains** (the Informix engine and the host operating system), reflecting the hard truth that a misconfigured or insecure host undermines every control the engine has. Within each domain are **categories**; security is represented as a first-class category in `both` layers (`IX_SECURITY`, `OS_SECURITY`) rather than as a separate silo. The catalog totals **161 Informix engine probes and 47 shared host probes: 208 in all**, spanning **19 categories**, each classified by category and kind (SQL against the catalogs, an ONSTAT command, or a host OS command).

Table 6 — The diagnostic categories, by domain

Domain	Category (display label)	Probes
Informix engine	Configuration	19
	CPU	8
	Memory	14
	Logs & recovery	11
	Disk performance	18
	Disk space	18
	Sessions & locks	21
	Users & applications	4
	SQL profiling (trace)	9
	SQL active (live queries)	5
	Network (Informix subsystem)	3
	Security	7
	Backup	9
	Replication & HA	15
Operating system	OS / system	12
	OS configuration	15
	Host network	7
	Host storage	8
	Host security	5

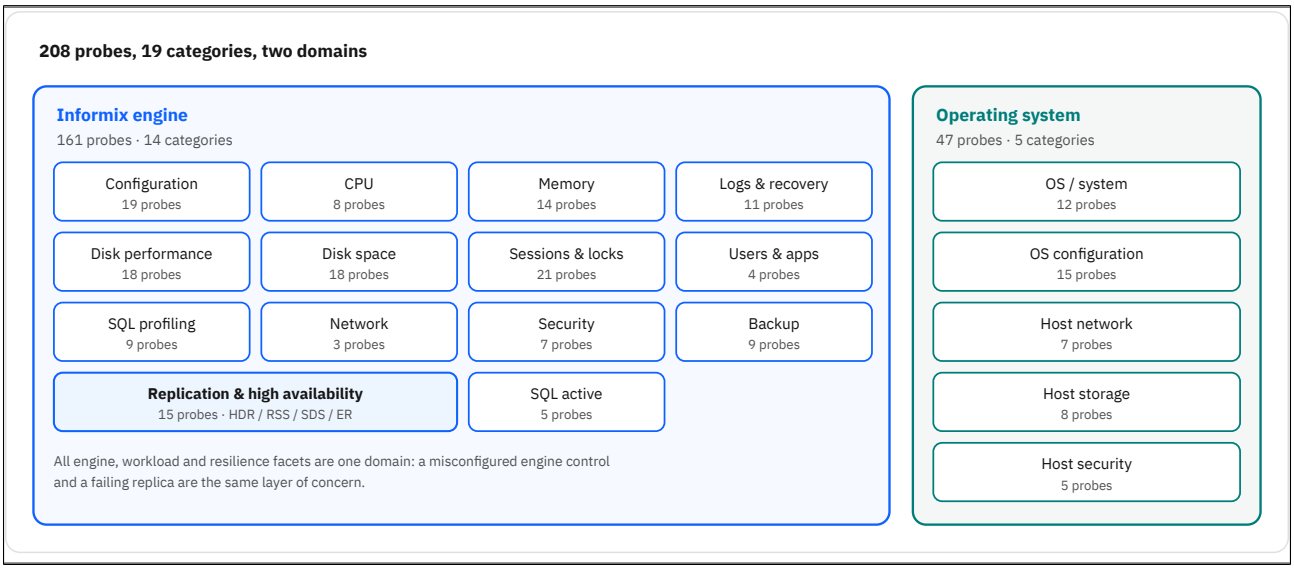

Figure 4 — The two-level coverage taxonomy: 19 diagnostic categories across two domains, the Informix engine and the host operating system

Informix NETWORK vs OS_NETWORK: not the same layer

The catalog draws a distinction that matters on a replicated estate. **Network** (`IX_NETWORK`) is `Informix's own` network subsystem read from `sysmaster` : NETTYPE buffer pools, poll threads, per-client session I/O. **Host network** (`OS_NETWORK`) is the `host` NIC and TCP stack: NIC error/drop counters, listen backlog, conntrack. A NIC silently dropping frames explains connection timeouts and HDR/ER lag that the engine's own counters cannot; keeping the two layers distinct is what lets the synthesis engine cascade a host-network fault up into an Informix-network symptom.

5.1. Engine coverage: the probe catalog in full

5.1.1. Configuration (19): the architecture-assessment layer

The parameters, optimizer-statistics health and schema choices that shape every other behaviour. Most "performance" problems on a mature estate begin here, in a value left at its install default or statistics gone stale, not in the workload itself. **Key output:** parameters that differ from their defaults, tables with missing, weak or old statistics, and schema risks such as extent explosion and cross-database dependencies.

Table 7 — IX_CONFIGURATION probes

Probe	Kind	What it checks
ix.cfg.nondefault	sql	Parameters differing from their ONCONFIG default.
ix.cfg.onconfig	onstat	The active ONCONFIG as the engine sees it.
ix.cfg.max_fill_data_pages	sql	MAX_FILL_DATA_PAGES setting.
ix.integrity.fk_constraints	sql	Unenforced foreign-key constraints.
ix.cfg.pfsc	sql	Partition Free-Space Cache configuration.
ix.schema.extents	sql	Table extent explosion (fragmentation).

Probe	Kind	What it checks
ix.schema.synonyms	sql	Synonyms resolving to a table in another database or server, the cross-DB dependency map.
ix.cfg.fot_indexes	onstat	Indexes suffering root-node (forest-of-trees) contention, from the buffer-mutex spin-lock hot spots.
ix.db.locale	sql	Database locale / collation coherence.
ix.cfg.stats_accuracy	sql	Estimated vs actual rows, the stale-statistics signature.
ix.cfg.stats_backlog	sql	Pending UPDATE STATISTICS (AUS backlog).
ix.cfg.autostats	sql	AUTO_STAT_MODE / STATCHANGE auto-maintenance posture.
ix.cfg.stats_dist	sql	Weak column distributions.
ix.cfg.index_usage	sql	Index read-vs-write, drop candidates.
ix.cfg.stats_fresh	sql	When LOW statistics were last built.
ix.cfg.stats_nodist	sql	Tables with no column distributions.
ix.sched.tasks	sql	Scheduler tasks and sensors, and their status.

5.1.2. CPU (8)

Where the engine's processor time actually goes, from virtual-processor classes down to the individual shared-memory spin locks. High CPU is almost always a symptom, so the value here is telling a genuine compute shortage apart from a bad plan, a session storm or lock contention wearing a CPU costume. **Key output:** VP CPU and ready-queue depth by class, the hottest spin locks and the objects they contend on, and host load.

Table 8 — IX_CPU probes

Probe	Kind	What it checks
ix.cpu.vps	sql	Virtual processors: CPU use and ready queue by class.
ix.cpu.glo	onstat	MT global / VP CPU usage.
ix.cpu.rea	onstat	Ready (runnable) threads.
ix.cpu.act	onstat	Active threads.
ix.cpu.spinlocks	onstat	Spin locks with waits, shared-memory contention.
ix.cpu.hot_partitions	onstat	Resolves the partition-manager spin locks to the actual objects contended on.
ix.cpu.os_load	os	Host load average and CPU count (Linux + AIX specs).

Probe	Kind	What it checks
ix.cpu.os_top	os	Top host processes by CPU (Linux + AIX specs).

5.1.3. Memory (14)

How the engine uses shared memory, from the buffer-pool caches down to the hard SHMTOTAL ceiling. It matters because Informix cannot allocate past SHMTOTAL: exhaustion means failed allocations and crashes, not merely slowness. **Key output:** buffer-pool hit rates per page size, memory by segment class, PDQ and statement-memory grants, and the remaining headroom to the SHMTOTAL cap.

Table 9 — IX_MEMORY probes

Probe	Kind	What it checks
ix.mem.allocation	sql	Informix memory by segment class.
ix.mem.mem	onstat	Memory pools.
ix.mem.mgm	onstat	Memory Grant Manager: PDQ / statement memory.
ix.mem.paging	os	Active paging rate (Linux).
ix.mem.os_free	os	Host memory and swap (Linux + AIX specs).
ix.mem.os_swap	os	What is swapped out, per process (Linux + AIX specs).
ix.mem.os_topmem	os	Top host processes by memory (Linux + AIX specs).
ix.mem.profile	onstat	Server profile: read/write cache ratios.
ix.mem.pools	sql	Buffer pools by page size, independent hit rates.
ix.mem.bufcoverage	sql	Whether each page size has a buffer pool; a dbspace whose page size has none cannot cache.
ix.mem.bufturnover	sql	Buffer-pool turnover ratio.
ix.mem.shmtotal	sql	**SHMTOTAL headroom**: the hard cap on the engine's total shared memory, and how close it is.
ix.mem.seg	onstat	Shared-memory segments.

5.1.4. Logs & recovery (11)

The physical and logical logs, checkpoints and the message log: the machinery that keeps the instance recoverable and its transactions durable. It matters because an undersized or saturated log stalls transactions, and a mis-sized log or a long transaction can freeze the entire server — the Blocked: CKPT state where the engine blocks every transaction to let a checkpoint finish before the physical log overflows, or the long-transaction "emergency brake" (LTXEHWM) that suspends all user threads while one runaway transaction rolls back. The four physical-log, checkpoint and long-transaction knobs are read as one connected subsystem: sizing is judged not by a static ratio but by the engine's own week-long checkpoint

history (sysadmin:mon_checkpoint), and every proposed value is justified inline with the verbatim IBM rule and a link to the source. **Key output:** a single “log & transaction sizing” posture (each parameter scored OK / review / action, watermarks translated from percentages into GB), why checkpoints fire and whether they block, the online.log tail, and any assertion-failure dumps.

Table 10 — IX_LOGS probes

Probe	Kind	What it checks
ix.log.plog_sizing	sql	Physical-log fullness and sizing (forced-checkpoint / recovery-cushion advisor).
ix.log.checkpoint_causes	sql	What causes checkpoints: trigger mix (colour-coded) and blocking.
ix.hist.checkpoints	sql	Checkpoint history from the engine sensor: plog/llog pages and combined MB per checkpoint.
ix.log.logical	sql	Logical-log status, with the LTXHWM/ LTXEHWM watermarks marked (in GB) on the log-space strip.
ix.log.switch_rate	sql	Logical-log switch cadence (last 24 h) vs IBM's 15-minute floor.
ix.log.fillrate	sql	Logical-log fill rate (last 7 days).
ix.log.message	onstat	Message-log tail (online.log).
ix.log.dump_readiness	os	Crash-dump capture readiness.
ix.af.list	os	Assertion-failure files in DUMPDIR.
ix.cfg.db_logging	sql	Per-database logging mode.
ix.sched.alerts	sql	Task Scheduler alerts (RED/YELLOW colour-coded).

5.1.5. Disk performance (18)

How efficiently the engine moves pages between disk and memory: chunk service times, page cleaning, read-ahead and the scans that drive I/O. A "slow disk" complaint is usually a plan reading far more than it should, so this layer separates real device latency from wasteful I/O. **Key output:** per-chunk service times, AIO and I/O-queue depths, page-cleaner and read-ahead effectiveness, and the tables taking the most sequential scans.

Table 11 — IX_DISK_PERFORMANCE probes

Probe	Kind	What it checks
ix.io.aiovp	onstat	AIO virtual-processor activity.
ix.io.checkpoints	onstat	Checkpoint history and timings.
ix.io.chunks	onstat	Chunk read/write counts.
ix.io.chunksvc	sql	Per-chunk I/O service times (ms).
ix.hist.activity	sql	Engine activity history (sensor).
ix.io.counters	sql	Key I/O and contention ratios.

Probe	Kind	What it checks
ix.io.extents	sql	Most-fragmented tables by extent count.
ix.io.pfsc	onstat	Partition Free-Space Cache effectiveness.
ix.io.fgwrites	onstat	Page-cleaning / foreground writes.
ix.io.iof	onstat	Async I/O by chunk / file.
ix.io.ioq	onstat	I/O queue depths.
ix.io.bufwaits	onstat	Buffer headers with a thread parked waiting, threads serialising on a hot page (onstat -X).
ix.io.lru	onstat	Buffer-pool LRU queues (dirty pages).
ix.io.direct_io	os	DIRECT_IO on cooked-file chunks (Linux + AIX specs).
ix.io.scans	onstat	Active scans: light / sequential / index.
ix.io.tablescans	sql	Tables with the most sequential scans.
ix.io.busytables	sql	Busiest tables by physical I/O.
ix.io.readahead	sql	Read-ahead effectiveness.

5.1.6. Disk space (18)

Where storage stands across dbspaces, sbspaces and chunks, and which tables consume it. It matters because a full dbspace stops writes outright, and sbspace metadata saturation can corrupt smart large objects. **Key output:** dbspace and sbspace utilisation (auto-expand-aware), per-chunk free space and availability, and the largest tables by size and by row count.

Table 12 — IX_DISK_SPACE probes

Probe	Kind	What it checks
ix.space.dbspaces	onstat	Dbspaces and chunks.
ix.space.free	sql	Dbspace utilisation (%), auto-expand-aware.
ix.space.os_df	os	Host filesystem usage (Linux + AIX specs).
ix.space.sbspaces	sql	Sbspace utilisation (%).
ix.space.temp_contents	sql	Temporary objects and who created them.
ix.space.temp_dbspaces	sql	Temporary dbspace provisioning.
ix.space.table_density	sql	Table page density, wasted space.
ix.space.toptables	sql	Largest tables: rows and size, partition-page cap.
ix.space.toptables_byrows	sql	Largest tables ranked by row count , the growth-driver view that complements ranking by pages.

Probe	Kind	What it checks
ix.space.layout	os	Chunks by filesystem, device and file permissions.
ix.space.chunkmap	sql	Every chunk's size and free space per dbspace.
ix.space.chunk_status	sql	Chunk availability (offline / inconsistent / recovering).
ix.space.databases	sql	Database sizes.
ix.space.sbspace_columns	sql	Smart-large-object (BLOB/CLOB) columns.

5.1.7. Sessions & locks (21)

The concurrency layer: who is connected, what they are running, and where they block each other. It matters because a lock-table storm or a single long-held lock can freeze an entire application tier in seconds, and the culprit is rarely the session that complains. **Key output:** the live session directory, who blocks whom, lock- and transaction-table pressure, and the heaviest sessions by I/O, memory and lock waits.

Table 13 — IX_SESSIONS_LOCKS probes: the concurrency layer

Probe	Kind	What it checks
ix.ses.directory	sql	Point-in-time directory of active sessions keyed by `sid` : state (running / waiting / idle), owner and program. The stable anchor the other session probes link to.
ix.lock.locks	onstat	Locks held / waited.
ix.lock.overflow	sql	Lock / transaction table overflows.
ix.lock.pagelevel	sql	Tables still using page-level locking.
ix.lock.waiters	onstat	Waiting threads.
ix.lock.lmx	onstat	Locked mutexes, internal contention.
ix.tab.open_partitions	onstat	Open table partitions.
ix.ses.idle	sql	Idle client sessions.
ix.sessions.by_database	sql	Sessions attached per database.
ix.ses.blockers	sql	Blocking sessions: who blocks whom.
ix.lock.contended	sql	Contended tables: lock mode vs contention.
ix.ses.heavy	sql	Heaviest sessions (I/O, scans, sorts, lock waits).
ix.ses.lockholders	sql	Top lock holders.
ix.ses.memory	sql	Top sessions by memory.
ix.ses.sql	onstat	SQL state per session (type, isolation, lock mode).
ix.ses.users	onstat	User / session threads.

Probe	Kind	What it checks
ix.ses.waiting	sql	Sessions in a wait state.
ix.ses.waits	sql	Wait breakdown: where the engine's waits go.
ix.tx.transactions	onstat	Transactions.
ix.tx.pinning	sql	Transactions pinning logical logs.
ix.hist.locks	sql	Lock / latch history (sensor).

5.1.8. Users & applications (4)

This category and the two that follow are the workload-attribution and statement-forensics layer: the answer to " who is doing this and with which statement", including for sessions that have already disconnected (the SQL-trace window survives them).

Table 14 — IX_USERS: per-user / per-application attribution

Probe	Kind	What it attributes
ix.usr.top	sql	Heaviest connected users right now.
ix.usr.apps	sql	Heaviest applications, by client program (feprogram).
ix.usr.cpu	sql	Top users by CPU seconds across all their session threads.
ix.usr.sql	sql	Top users by traced statement load (survives disconnect).

5.1.9. SQL profiling — trace-based (9)

Statement-level forensics from the SQL-trace ring: the costliest statements and the plans they actually ran, including for sessions that have already disconnected (the trace window survives them). This is the highest-value diagnostic on the engine and the hardest to capture safely, for reasons the dedicated section that follows sets out. Because the trace records a statement only on completion, the companion SQL active category catches queries that are still running. **Key output:** top statements by execution time, reads and disk sorts; the sequential-scan and misestimate signatures of a bad plan, with the indexes that plan could have used; and the executed plan tree of a completed query.

Table 15 — IX_SQL_PROFILING: statement and executed-plan forensics

Probe	Kind	What it profiles
ix.stmt.trace_status	sql	Whether the SQL-trace ring is capturing (gates the rest).
ix.stmt.toptime	sql	Biggest cumulative time sinks (frequent + slow).
ix.stmt.slowest	sql	Slowest single executions.
ix.stmt.topreads	sql	Top statements by physical disk reads.
ix.stmt.disksorts	sql	Statements spilling sorts to disk.

Probe	Kind	What it profiles
ix.stmt.seqscans	sql	Plans that sequentially scanned a large table.
ix.stmt.scan_indexes	sql	The indexes available on the sequentially-scanned tables — was a better plan possible?
ix.stmt.misestimate	sql	Actual rows >= 10x the estimate, stale-stats signature.
ix.stmt.crossdb	sql	Traced statements whose executed plan touches more than one database, the cross-DB workload that is often the real root cause.

5.1.10. SQL active — live running queries (5)

The SQL trace records a statement only on `completion`, so a still-running monster query is invisible there. This category catches it `while it runs`: the most expensive query executing right now, its live plan operators with rows processed so far, how long it has been running, and the temporary space it is consuming. These read the running state directly, so they work even with tracing off. **Key output:** the costliest running statement, live plan operators, elapsed-time ranking, and per-session temporary-space usage.

Table 16 — IX_SQL_ACTIVE: queries executing right now

Probe	Kind	What it checks
ix.stmt.cost	sql	Most expensive `running` SQL by optimizer cost (works with tracing off).
ix.stmt.longrunning	sql	Statements executing now, ranked by elapsed time.
ix.ses.pqs	onstat	Live query operators of running plans, with the rows processed so far.
ix.stmt.tempspace	sql	Top sessions by temporary space (sorts, hashes, SELECT INTO TEMP).
ix.stmt.tempobjects	sql	The temporary objects held per session, behind the temp-space total.

5.1.11. Network (3)

Informix's `own` network subsystem, read from `sysmaster`: NETTYPE buffer pools, poll threads and per-client session I/O. It matters because poll-thread or buffer starvation surfaces as mysterious client timeouts and is easily mistaken for a host or application fault (the host NIC layer is a separate category). **Key output:** NETTYPE tuning counters, connections allowed, accepted and rejected, and the chattiest client sessions by bytes.

Table 17 — IX_NETWORK: Informix's own network subsystem

Probe	Kind	What it reads
ix.net.global	sql	NETTYPE buffer-pool tuning counters and network control-block usage.

Probe	Kind	What it reads
ix.net.clients	sql	Connections allowed / accepted / rejected and I/O per client type.
ix.net.io	sql	Per-session network read/write bytes, the chattiest connections.

5.1.12. Backup (9)

The backup category reflects a hard-won operational truth: **backup incidents are overwhelmingly control-plane failures**: a device pointing at `/dev/null`, a missing `DSMI_CONFIG`, a stale NFS mount, a dbspace never archived, not engine faults. The probes therefore inspect the plumbing, not just the catalog.

Table 18 — IX_BACKUP probes

Probe	Kind	What it verifies
ix.cfg.recoverability	sql	Archive / log-backup device settings; `/dev/null` = no backup possible.
ix.backup.spaces	sql	Time since last level-0/1/2 archive per dbspace.
ix.backup.history	sql	ON-Bar action history from the sysutils catalog (incl. failures).
ix.backup.device	os	Where TAPEDEV/LTAPEDEV actually point, with free space.
ix.backup.barlog	os	ON-Bar / XBSA / PSM layer where a failure shows its REASON.
ix.backup.timing	os	Archive events and durations from online.log (the ontape record).
ix.backup.host_readiness	os	Host-side gaps that fail backups silently (DSMI, NFS, permissions).

5.1.13. Security (7)

The engine's security posture: privilege breadth, listener and replication encryption, auditing and known-CVE exposure. It matters independently of performance, because a database granted broad `PUBLIC` rights or listening in plaintext is a standing breach waiting to happen. **Key output:** over-broad grants, plaintext-TCP and unencrypted-replication exposure, auditing state, passwordless host trust, and the engine's CVE and fixpack gap.

Table 19 — IX_SECURITY probes

Probe	Kind	What it assesses
ix.sec.grants	sql	Over-broad privileges (PUBLIC / DBA).
ix.sec.dumpshmem	sql	Shared-memory credential dumps (DUMPSHMEM).
ix.sec.cve_gap	sql	Engine level vs known security fixpacks.
ix.sec.tls	os	Listener encryption and plaintext-TCP exposure.

Probe	Kind	What it assesses
ix.sec.replication_crypto	os	HDR / RSS / SDS / ER stream encryption.
ix.sec.audit	os	Database auditing (onaudit / ADTMODE).
ix.sec.sqlhosts_trust	os	sqlhosts passwordless trust (s= option).

6. SQL Statement Profiling: the Hardest Evidence to Capture Safely

On a large estate the single bad statement is the most common root cause hiding behind a CPU spike, a "slow disk" complaint or a lock storm. Naming that statement, and the plan it `actually` ran, is the highest-value question in diagnosis. It is also the hardest to answer safely, because the only complete source of that evidence, the engine's SQL trace, is both expensive to run and treacherous to read. This is precisely the kind of sophisticated capability that is difficult to operate by hand, and getting it wrong has consequences.

Two ways naive statement profiling backfires

The memory problem. Turning SQL tracing on is not free. Its buffers live in the engine's shared memory, drawn from the same fixed `SHMTOTAL` budget the whole instance runs under. Sized carelessly on a busy system, the very act of trying to profile a stressed engine can push it toward memory exhaustion: the instrument becomes the outage. And it is exactly the largest, busiest instances, the ones closest to their memory ceiling, that most need profiling and least tolerate a careless "just turn it on".

The ring problem. The trace is a finite ring buffer. On a busy engine it wraps in `seconds`, so the statements that caused an incident are overwritten before anyone connects to look. Worse, the ring keeps moving while it is read: a statement can be evicted between the moment its cost is ranked and the moment its text or plan is fetched, so a careless reader gets a confident ranking of the `wrong` statements, or an execution plan shown under a statement that never ran it. The evidence does not merely vanish; it can quietly mislead.

The agent makes statement profiling both **safe** and **trustworthy**, so the operator never has to weigh that risk. It:

- never lets tracing threaten the engine's memory, standing down rather than pushing an already-stressed instance past its ceiling;
- captures a single coherent, frozen view, so every ranking, fingerprint and executed plan describes the same moment instead of a shifting ring;
- knows how much history it actually captured, and says so, warning when the window was too short for a ranking to be trusted;
- and surfaces the outcome the way a DBA wants it: the costliest statements by real execution time, each tied to the plan it truly ran.

The result is statement-level forensics you can act on, captured safely on exactly the systems where it was previously too risky to collect. The engineering that delivers this, the isolated capture, the memory budgeting and the pre-eviction snapshot, is detailed under the collection model earlier in this document; here it is enough to say that the hard problem is solved and the operator sees only the answer. Console captures of the profiling grid and an executed plan appear in the console appendix.

7. The Second Modality: Continuous Observation on the Wire

Everything to this point describes one way of watching a system: the diagnostic agent reaches in over a single JDBC connection, **periodically**, collects structured evidence and reasons from symptom to cause. There is a second, complementary way, and the platform now ships it as a **distinct product — a second flavor of the same agent**: a passive **wire-tap** that never reaches in at all. Where the diagnostic agent **diagnoses** — periodically, broadly, analytically — the wire-tap **observes**: continuously, in real time, at the level of the individual SQL statement as it crosses the wire. The two answer different questions. One tells you `what is wrong and why`; the other tells you `what is happening, right now, and what it is costing`.

The wire-tap runs **on the database host** (or on a span port) and does exactly one thing: it reads copies of the packets on the Informix listener port off the network interface, reassembles each TCP connection, and decodes the SQLI protocol the client and server already speak. From that stream alone it reconstructs every statement — its text, its **client-observed latency** (split into server think-time and result-transfer time), its result rows and bytes on the wire, and its error code — and groups value-bearing statements by a literal-stripped fingerprint into one logical unit. It holds exactly one operating-system capability, `CAP_NET_RAW`. It **never opens a database connection**, never authenticates, never writes a byte to the wire, and adds **no load to the engine**: no trace buffer to enable, no ring to pressure, no sbspace, nothing to turn on or off on a box that is already close to tipping over. It is non-interference taken to its limit — the observer is not merely light, it is not in the conversation at all.

> **It closes the structural blind spots of the trace ring.** The engine's own SQL trace is indispensable but records a statement only on `completion`, keyed to a session, in a ring that wraps. The wire-tap sees the same traffic differently:
>
> - **The still-running statement.** The trace ring records nothing until a statement finishes, so the monster query holding the system hostage is invisible there while it does the damage. The tap sees the request the instant it hits the wire and holds it, elapsed growing, until it completes — the long-runner is caught `during` the incident, not reconstructed after.
> - **The real client.** The engine attributes traffic poorly by client host; the tap keys every statement to the true client IP from the TCP tuple.
> - **The disconnected session.** A session's per-statement counters die with it; the tap has already captured them off the wire.

The operator watches this on a live, theme-aware console updated in near-real-time: the client **connections** in flight (with open-cursor-leak detection), the **transactions** — each `BEGIN→COMMIT` unit grouping the statements that ran inside it — the **top statements** by cumulative time, the queries **executing right now** with live-growing elapsed, and **by-client** attribution covering even sessions that have already gone. A top-level manual documents every grid and the protocol behind it; a headless mode writes the same capture to a self-contained report for offline reading.

One table survives a restart, and it is the one that earns the wire-tap its place as a monitoring instrument rather than a live curiosity: a **Top history** of the heaviest SQL, accumulated across every session the tap has run. Here it does the thing an average hides — it **separates the two failure modes that both matter and look nothing alike**. A fast statement run millions of times accumulates enormous total time: the aggregate hog, often the biggest prize and the hardest to fix, because it is the application's fundamental access pattern. A single statement that runs for minutes but only occasionally barely registers on total time, yet it is individually ruinous — and usually one index or one rewrite away. Ranked by cumulative time by default, the table **re-ranks on demand by maximum, by average, or by frequency**, and its retention deliberately protects the leaders of `each` axis, so the slow-but-rare query is never quietly evicted by a total-time cap. A class badge flags the

dangerous middle the two extremes conceal: the two-second statement run so often that it dominates both the total time and the concurrency, silently, below every per-statement alert threshold.

> **Observation feeds diagnosis.** The two flavors are one platform. On request, the wire-tap exports a self-contained bundle that the console ingests through the `same` pipeline as any offline collection — a `source=WIRE` triage session — so wire-observed heavy SQL enters the same reasoning engine, the same reports and the same worst-consumer ranking as engine-collected evidence. Run the diagnostic agent to **understand** the estate at a moment; run the wire-tap to **watch** it continuously; and let a long-runner the tap catches become, without a human in the loop, evidence in the next diagnosis.

8. Replication and High-Availability Intelligence

For a large estate the HA topology is not a feature to be checked: it is the substrate everything else runs on, and it fails `silently`. In practice, replication breaks in characteristic ways: HDR and RSS secondaries drop, apply lag grows, an SMX connection stops, a log wrap defeats resynchronisation, ER queues back up behind a blocked apply thread, or the `syscdr` catalog goes stale. A replicated estate that is not actively watching these signals is, in effect, operating its high-availability layer **blind**. The agent treats HA/ER as a first-class diagnostic domain with dedicated probes, rules, an architect narrative and defect matching.

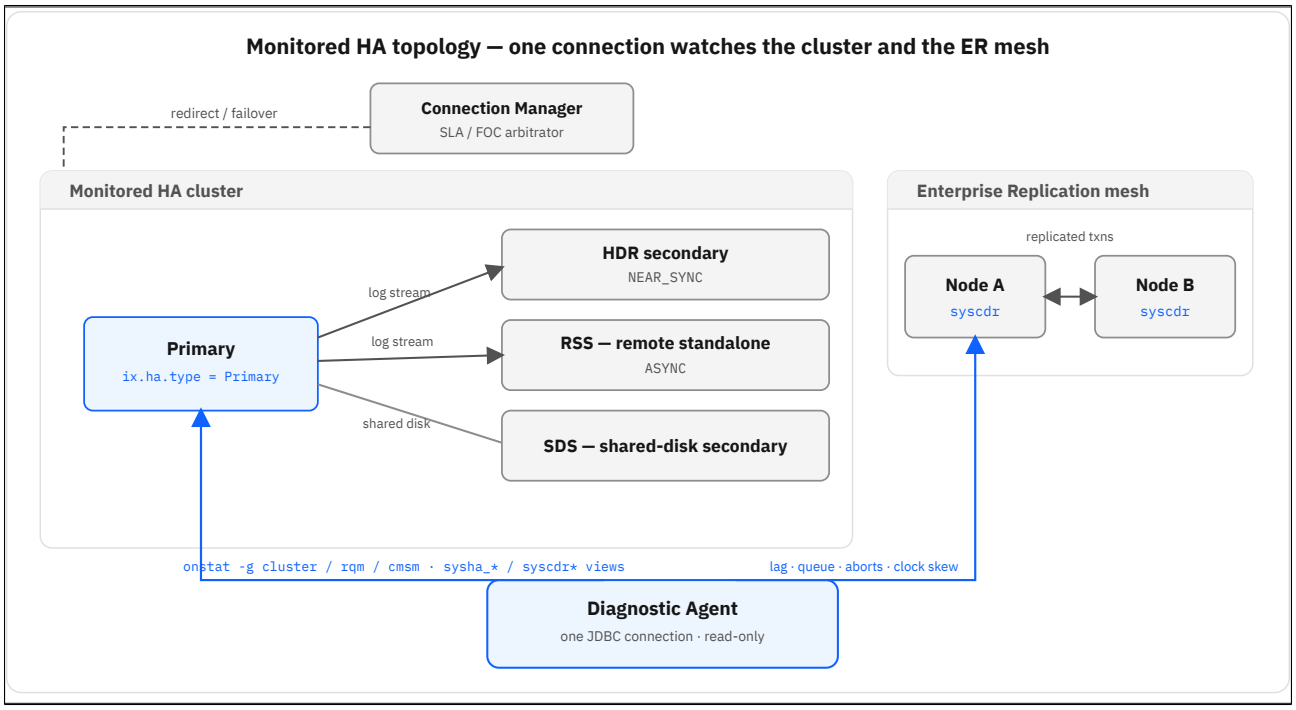

Figure 5 — Monitored HA topology: an HDR/RSS/SDS cluster and an Enterprise Replication mesh observed through one connection

Table 20 — Replication & HA coverage: probes (IX_HA)

Probe	Kind	Signal it raises
`ix.ha.type`	sql	Instance role (Standard / Primary / HDR / SDS / RSS secondary) from `sysha_type`.
`ix.ha.dri`	onstat	`onstat -g dri`: replication state from `this` node's own point of view (DR type and state).
`ix.cfg.secondary`	sql	The ONCONFIG parameters that shape secondary behaviour (HDR/RSS/ SDS delayed-apply, index shipping, updatability).
`ix.ha.nodes`	sql	Every paired server and its type / state (`sysha_nodes`).
`ix.ha.lag`	sql	Per-secondary apply lag (`sysha_lagtime`); alerting signal warn 5 s / crit 30 s.
`ix.ha.cluster`	onstat	`onstat -g cluster`: each secondary's sync type and Connected/Active state.

Probe	Kind	Signal it raises
ix.ha.rss	onstat	onstat -g rss : remote-standalone-secondary state, index-page logging and flow control.
ix.ha.sds	onstat	onstat -g sds : shared-disk-secondary state and server count.
ix.ha.smx	onstat	onstat -g smx : the Server Multiplexer that carries all RSS and SDS traffic; a silent SMX stop.
ix.ha.proxy	onstat	onstat -g proxy : the distributor forwarding writes from an updatable secondary to the primary.
ix.er.nif	sql	ER peer interface connections; a disconnected interface stalls replication.
ix.er.tx	sql	Replicated transaction counts and aborts (aborts = targets diverging).
ix.er.rqm	onstat	onstat -g rqm : ER send/receive/control queues spilling to sbspace.
ix.cm.sla	sql	Connection Manager units and their SLA redirection state.
ix.cm.detail	onstat	onstat -g cmsm : CMs, SLAs, failover arbitrator; guards against split-brain / no-redirect.

The structured metrics behind these probes are computed, not scraped. The HA role (drRole) and whether a secondary is attached (drHasSecondary) come from sysha_type ; collectHaTopology() runs onstat -g cluster , joins the logical-log geometry from syslogs , and expresses each secondary's lag **in pages** against the primary's current log position. That lets the rules judge lag as a fraction of the cluster's log capacity, the number that actually predicts a lose-sync event.

Table 21 — Replication & HA coverage: rules and reasoning

Mechanism	Behaviour
replicationLagRisk	Secondary apply lag >= 25% (warning) / 50% (critical) of logical-log capacity, **exempting a secondary deliberately configured for delayed apply** (DELAY_APPLY / STOP_APPLY), so an intentional delay is not misread as a fault.
noDisasterRecovery	A production primary/standalone with **no** HA or RSS secondary at all: the estate has no disaster-recovery partner.
logIndexBuildsOffWithRss	LOG_INDEX_BUILDS off while an RSS secondary is attached: index pages are not shipped, so the RSS silently rebuilds them and can fall behind.
hdrManualWithoutCm	HDR in manual failover with no Connection Manager arbitrating: a failover has no automated redirect.

Mechanism	Behaviour
`erQueueMemUndersized`	ER configured while CDR_QUEUEMEM sits at its 4 MB default: the send queue will spill.
`logReplicationEvents`	DR or SMX events crossing threshold in the online.log tail.
`bufferedLoggingOnHaPrimary`	Buffered logging on an HA primary with a connected secondary (engine error 188).
Blocked-state `DDR` classification	A `DDRBLOCK` engine state (ER/HDR log processing fell behind) is routed to the HA category with a specific playbook.
`ix.sec.replication_crypto`	HDR/RSS/SDS active with `ENCRYPT_HDR=0` is a critical finding; ER active with `ENCRYPT_CDR=0` a warning.
`os.clock`	ER with timestamp / delete-wins conflict resolution requires clocks synced to +/-1 s and never stepping backward, IBM's one hard clock rule. Drift silently corrupts conflict resolution.
Architect verdict (`haVerdict`)	Narrates the cluster: standalone ("no failover partner exists"), a secondary ("this node is the safety net"), a degraded cluster (a peer not Connected), or a protecting cluster with the worst lag as a % of log space.

Per-mode replication depth: now landed

The ONSTAT replication captures now span the full per-mode set (`-g cluster` , `-g dri` , `-g rss` , `-g sds` , `-g smx` , `-g proxy` , `-g rqm` and `-g cmsm`), complemented by the `sysha_*` and `syscdr*` catalog views and the secondary-shaping ONCONFIG parameters (`ix.cfg.secondary`). The finer per-mode flags that a previous release deferred to remediation guidance are now **dedicated probes**; what remains on the roadmap is deeper ER apply-thread introspection.

The value proposition was never exhaustive flag coverage. It is that lag, queue backlog, peer disconnection, a silent SMX stop, encryption gaps and clock skew are watched **continuously and judged against thresholds**, which is precisely what "operated blind" was the absence of.

Diagnose the secondary directly, read-only and all

A replicated estate must often be inspected on a **read-only secondary**, where an HDR, RSS or SDS server accepts no writes and many ordinary catalog operations behave differently. The agent detects the server's replication state at connection time and adapts, running its **full** diagnosis against a read-only secondary rather than failing or silently degrading to a partial one. This matters operationally: it lets a site point the agent at a standby, off the primary's critical path, and still obtain the complete picture, including that node's own view of the cluster.

9. Dual-Platform Host Layer: Linux and AIX/POWER at Parity

A large Informix estate is very often an **AIX/POWER** estate — IBM's own enterprise platform, where the biggest and most business-critical Informix systems have run for decades — and AIX is precisely where generic and third-party monitoring gives up. There is no text `/proc`; `/proc/<pid>/status` is a binary struct; and the facts live in platform tools (`svmon`, `lsps`, `lparstat`, `vmstat -v`) that most host exporters never learned, so weak AIX coverage is the single most common blind spot in the tooling an Informix shop can otherwise buy. This agent instead treats the host as a **peer diagnostic layer at full Linux/AIX parity** — not a degraded fallback mode — with **47 host probes** whose per-platform command specs resolve to the right source on each operating system. Delivering the same depth on AIX/POWER as on Linux, from one unprivileged JDBC connection, is a core differentiator: it is exactly the platform where an estate most needs the help and is least likely to get it. The platform is detected once per connection via `uname -s`; the metrics collector branches to `collectHostAix()` or `collectHostLinux()` accordingly.

Table 22 — Host-fact sources: Linux vs AIX/POWER

Fact	Linux source	AIX source (4 KB pages)
Load average	`/proc/loadavg`	`uptime` (locale-pinned)
Logical CPU count	`nproc` / `/proc/cpuinfo`	the `lcpu=N` token in the `vmstat` header
RAM used / total	`/proc/meminfo` (MemAvailable)	`vmstat -v` (memory minus free minus file cache)
Swap / paging space	`/proc/meminfo` SwapTotal/Free	`lsps -s`
Informix swap slice	`/proc/[pid]/status` VmSwap	`svmon -P -O sortentity=pgsp` (Pgsp column)
Virtualization / entitlement	DMI / cgroup detection	`lparstat -i` (entitled capacity, capped/shared, pool CPUs) + `%entc` utilisation

Beyond the shared probes, the catalog carries platform-specific checks that have no counterpart on the other operating system:

- **AIX-only:** per-disk service time and queue (`os.hdisk`, `iostat -D`), the hardware error log (`os.errpt`), LVM mirror and physical-volume health (`os.lvm`), paging-space pressure (`os.paging`), memory-affinity SRAD placement (`os.srad`), VMM page-stealing tunables (`os.cfg.vmm`), and 16 MB large-page accounting (`os.cfg.largepages`).
- **Linux-only:** huge pages, THP, systemd RemoveIPC, `fs.protected_regular`, the block-device I/O scheduler, and software-RAID.

9.1. The host probe catalog: is Informix on a healthy platform?

Every host probe answers one question: **is Informix running on a healthy, correctly configured platform?** The engine's own tuning is only as good as the host beneath it — a kernel semaphore limit that caps connections, a paging space quietly filling, a mirrored disk that has silently lost redundancy, an end-of-support OS, or a NIC dropping packets will undermine every engine control. The 47 host probes read as the unprivileged `informix` user, grouped below by the four host domains; the **Platform** column shows where each applies (`Linux + AIX` probes resolve a per-OS command spec internally, so the same probe reads the right source on each operating system).

Table 23 — Host probes — System & runtime state (storage, CPU/memory pressure, virtualization, uptime)

Probe	Platform	What it checks
os.chunk_fs	Linux + AIX	Chunk filesystem safety (data-loss / defeated DIRECT_IO)
os.clock	Linux + AIX	Clock synchronisation (NTP / chrony)
os.cpu	Linux	Host CPU pressure — steal & I/O wait
os.cpu_bench	Linux + AIX	CPU speed benchmark (opt-in)
os.disk_bench	Linux + AIX	Disk speed benchmark (opt-in, read + write, O_DIRECT)
os.disk_paths	Linux + AIX	Storage path health (MPIO / multipath)
os.diskstats	Linux + AIX	Host disk I/O per device, judged by device class
os.errpt	AIX	AIX hardware error log (errpt)
os.filesystems	Linux + AIX	Filesystem health — read-only remounts & inode exhaustion
os.hdisk	AIX	AIX per-disk service time & queue (iostat -D)
os.limits	Linux + AIX	informix user resource limits
os.lvm	AIX	AIX LVM mirror & physical-volume health
os.mdraid	Linux	Linux software-RAID (md) inventory & health
os.oom	Linux + AIX	Out-of-memory kills
os.paging	AIX	AIX paging-space pressure & active page-out
os.srad	AIX	Memory affinity - SRAD placement and measured dispatch locality
os.storage_map	Linux + AIX	Storage topology — dbspace to physical disk & redundancy
os.uname	Linux + AIX	Kernel / OS release
os.uptime	Linux + AIX	Uptime & load
os.virt	Linux + AIX	Host platform — bare-metal, virtualized, or container

Table 24 — Host probes — OS configuration correctness (kernel limits, VM/paging, huge/large pages, KAIO, currency)

Probe	Platform	What it checks
os.cfg.aio	Linux	Kernel async-I/O request limit (KAIO)

Probe	Platform	What it checks
os.cfg.autostart	Linux	OS start/stop of the engine (systemd auto-start & clean shutdown)
os.cfg.cpufreq	Linux	CPU frequency governor (power management)
os.cfg.hugepages	Linux	Explicit huge pages (hugetlb pool)
os.cfg.io_sched	Linux	Block-device I/O scheduler (flash devices)
os.cfg.largepages	AIX	AIX 16 MB large pages - pool vs what oninit actually uses
os.cfg.numa	Linux	NUMA balancing & memory locality
os.cfg.os_currency	Linux + AIX	Operating-system currency (end-of-support / stale level)
os.cfg.protected_regular	Linux	fs.protected_regular breaks the SHM listener
os.cfg.removeipc	Linux	systemd RemoveIPC (instance-killer)
os.cfg.shm_sem	Linux	Kernel shared-memory & semaphore limits
os.cfg.thp	Linux	Transparent Huge Pages (THP)
os.cfg.ulimits	Linux + AIX	informix user resource limits (fsize / nofiles / maxuproc)
os.cfg.vm	Linux	Virtual-memory tunables (swappiness / overcommit / dirty)
os.cfg.vmm	AIX	AIX VMM page-stealing tunables

Table 25 — Host probes — Network (NIC health, TCP backlog, bonding/redundancy)

Probe	Platform	What it checks
os.net.bonding	Linux + AIX	NIC bonding / EtherChannel redundancy
os.net.errors	Linux + AIX	NIC errors, drops & link state
os.net.inventory	Linux + AIX	Host network-interface inventory (IP, MAC, MTU, bond membership)
os.net.link	Linux	NIC link speed & duplex
os.net.retrans	Linux	TCP retransmissions & listen-queue overflows
os.net.tcp	Linux + AIX	TCP listen backlog & connection tracking
os.net.throughput	Linux + AIX	NIC bandwidth utilisation (live)

Table 26 — Host probes — Security posture (permissions, trust, SSH, environment hardening)

Probe	Platform	What it checks
`os.file_perms`	Linux + AIX	Permissions on the engine's control & data files
`os.hardening`	Linux + AIX	informix environment hardening (umask / PATH / setuid)
`os.onsecurity`	Linux + AIX	Trusted-binary path security (onsecurity)
`os.ssh_config`	Linux + AIX	SSH daemon hardening & informix key hygiene
`os.trusted_hosts`	Linux + AIX	OS trusted-host authentication (rhosts / hosts.equiv)

The AIX gotchas that cost real debugging time are encoded once and for all: AIX `ps` spells RSS `rssize` and has no `--sort`; `lparstat` needs `ODMDIR` exported; and SMT threads are never mistaken for physical cores, the classic error behind an engine given far more CPU virtual processors than the LPAR has cores to dispatch.

Why POWER sizing is a first-class rule

On a shared-processor LPAR, the meaningful CPU headroom is entitlement, not thread count. The agent reads entitled capacity and measured dispatch locality, applies IBM's guidance (average utilisation should stay at or below ~75% of entitlement), and can flag an engine that has been given more virtual processors than the LPAR can physically dispatch, a misconfiguration that presents as "high CPU" but is really wasted context-switching. This is exactly the kind of standing, always-visible-never-watched risk that continuous diagnosis exists to surface.

10. The Reasoning Layer: From Metrics to a Ranked Diagnosis

Collection produces a structured `MetricsSnapshot`; the **diagnostic model** turns it into a ranked, narrated diagnosis. Unlike a large-language model, this model is **deterministic and auditable**: the same snapshot always yields the same findings, with the exact metric and threshold behind each one, whether evaluated live or re-analysed from a stored bundle six months later. That is the point: it delivers architect-grade reasoning you can `defend in a change-review`, not a plausible-sounding narrative you have to trust. The model has five cooperating parts: rules, synthesis, defect matching, a symptom-first guide, and a self-check that audits the finished diagnosis for internal contradictions before it is shown.

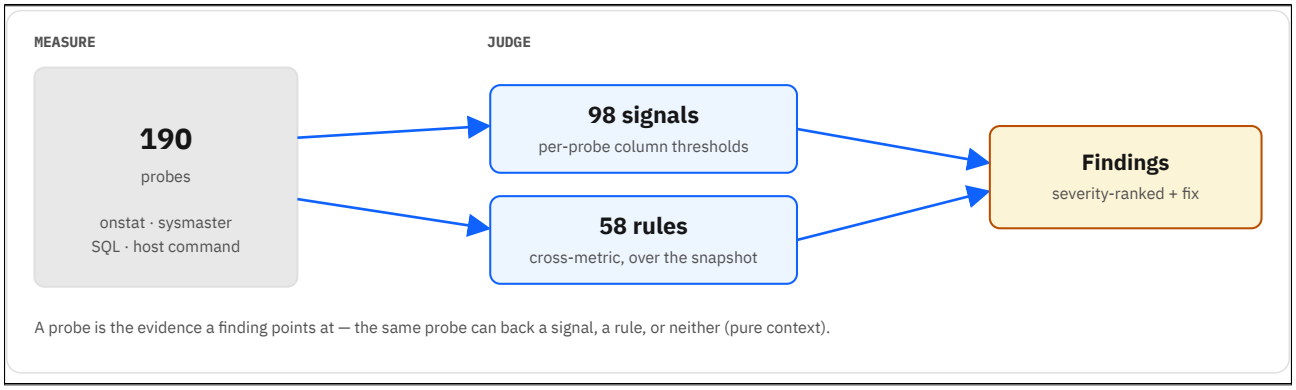

Figure 6 — The measure-then-judge model: probes feed signals and rules, which produce findings

10.1. The rule set

The model applies **53 snapshot rules and 5 interval-rate rules**, each emitting `Finding`s at one of three severities (**CRITICAL, WARNING, INFO**), sorted most-severe-first, then by category. Rules read named structured metrics; not one of them parses `onstat` text. A representative slice:

Table 27 — Representative rules by area (thresholds shown where illustrative)

Area	Rule	Fires when
Cache	`readCache` / `writeCache`	Read hit < 90/95%; write hit < 70/85% (suppressed while counters are young).
Host memory	`hostSwap`	Any swap in use; critical when `oninit` itself is paged out.
Shared-memory ceiling	`shmTotalNearLimit`	Allocated shared memory approaching the hard `SHMTOTAL` cap: the engine cannot allocate past it, so exhaustion means failed allocations, not just slowness.
Statement memory	`statementMemory` / `pdqGrantGating`	Sorts spilling to disk >= 5/20%; PDQ queries gated by the Memory Grant Manager.
Storage	`dbspaceFree` / `sbspaceFree` / `partitionPageCap`	>= 90/95% full (auto-expand and filesystem headroom chained in); a partition approaching the 16.7 M-page cap.

Area	Rule	Fires when
Storage integrity	`chunkFilePermissions`	A chunk file not `660 informix:informix`.
Checkpoints / logs	`checkpointBlocking` / `physicalLogTooSmall` / `logicalLogTooSmall` / `ltxWatermarkRisk`	Checkpoint blocking application threads; a physical/logical log too small for the workload; long-transaction high-water marks set to disaster.
Recoverability	`backupNotConfigured` / `unloggedDatabase` / `logicalLogSaturation`	TAPEDEV= `/dev/null`; a user DB with logging off; logs awaiting backup >= 75/90%.
Crash forensics	`assertFiles` / `logCrashSequence`	Recent assertion-failure dumps; a PANIC or fatal-trap-then-restart sequence in the log.
Concurrency	`lockWaits` / `deadlocks` / `waitBottleneck`	Lock-wait ratio, deadlock rate, or the share of engine wait time spent on locks.
Network	`networkWaits`	Network's share of the wait profile crossing threshold.
Statistics	`staleStatistics` / `statsAccuracy` / `slowStatementStaleStats`	AUS backlog with large estimate-vs-actual drift; a heavy traced statement over a stale-stats table.
Trace hygiene	`traceWindowTooShort`	The SQL-trace ring wrapped within seconds: trace rankings are not trustworthy.
Workload	`workloadConcentration`	One user/application dominating the engine (reported as INFO: attribution, not fault).
Interval rates	`seqScanRate` / `lockWaitRate` / `dominantStatement` / interval cache	Per-second rates over a sampling window, catching spikes a single snapshot averages away.

Fragile-state short-circuit

A snapshot taken on an engine that is `not` cleanly online would make ordinary rules misfire on missing zeros. The engine detects this from the low-overhead `onstat -` header and short-circuits to a single CRITICAL "diagnosis reduced" finding, routed to the exact cause: `LAST_LOG_RESERVED4BACKUP` to Backup, `DOWN_SPACE` to Disk space, `CKPT` to Logs, `DDR` to Replication, an archive block to Backup (citing APAR IC90129). The reduced finding is produced by one shared method, so the live limited path and an offline import agree exactly.

Built not to cry wolf

A monitor that alerts on noise is switched off, so the rules are engineered against false positives as deliberately as against misses:

- ratios computed from counters that have barely moved since engine start are **suppressed until the counters are old enough to mean something**;
- a replication secondary deliberately configured for delayed apply is **exempted** from the lag rule rather than flagged;
- a physical log carrying its own share of logging traffic is measured against its `real` workload, not a blanket threshold;
- and in continuous operation the alert stack fires **only on findings that are new** versus the previous cycle.

The intent is that when an alert does arrive, it is worth reading.

10.2. Causal synthesis: the architect's opinion

Findings are ranked, but ranking is not reasoning. The synthesis layer answers "**start here**" by assigning each finding a **causal rank** (how far upstream it sits), keyed on probe id with a category fallback. Host memory and CPU rank 0 (fix first); configuration, space and logs rank 1; buffer-pool memory rank 2; I/O and checkpoints rank 3; lock contention rank 4. It then assembles conservative **cross-layer themes** that fire only when at least two members co-occur:

- "Memory pressure is driving physical I/O" (memory + disk performance).
- "Weak optimizer statistics are producing scan-heavy plans" (stale stats + sequential scans).
- "Storage is approaching exhaustion" (two or more space / log-saturation findings).
- "This instance is running on build-template defaults": several provisioning parameters (physical and logical log sizing, `CDR_QUEUEMEM`, lock table) left at their install defaults co-occur, which points at an **untuned deployment** rather than a set of unrelated faults, and changes the recommendation from N piecemeal fixes to one "size this instance for its workload" action.
- Host-to-engine cascades: a host-network fault surfacing as an Informix-network symptom; host storage surfacing as engine disk performance.

The most-upstream root among the active themes becomes the **primary suspect**, the single thing to fix first. On top of this sits a registry of deterministic **architect verdicts** keyed by probe id (physical/logical log and its reusability, buffer pool, checkpoints, host memory, the `SHMTOTAL` shared-memory ceiling, VP topology and the CPU ready queue, PDQ memory-grant gating, dbspace, contention, statistics, read-ahead, the HDR data-replication posture and the wider HA cluster, the live running-operator view, backup, workload, and schema design). Each renders a short, plain-language narrative (`including a healthy one`), so the report reads like a senior architect walked the system, not like a threshold dashboard.

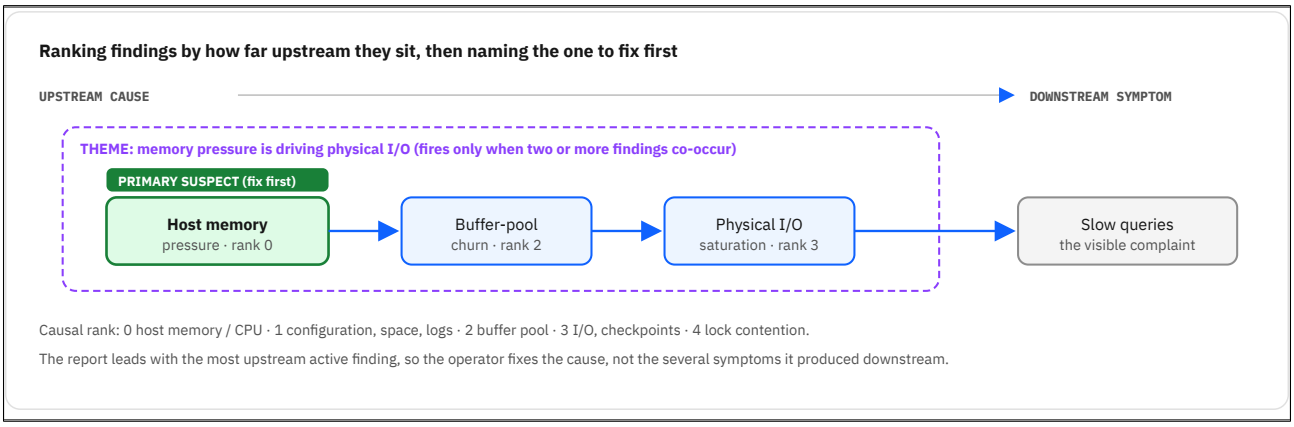

Figure 7 — From findings to a primary suspect: ranking each symptom by how far upstream it sits

10.3. Version-to-APAR knowledge base

A recurring and costly failure mode in long-lived estates is **re-encountering a known, already-fixed engine defect** after a migration or fixpack change. The agent carries a curated **version-to-APAR knowledge base** and matches it two ways:

Signature tier.
A verbatim `online.log` signature (a regex) matches the captured log tail `and` the running level falls in the affected fixpack range, a finding at the entry's own severity, naming the defect and the fix level. Entries include SMX log-xmit stalls, HDR-secondary rollforward panics after TRUNCATE, and ABENDs under ER load on a full CDR sbspace.

Exposure tier.
The running level has documented defects but none has fired in the log yet: one INFO listing each id, title and fix level, so the operator sees the latent exposure before it bites.

This turns tribal memory ("wasn't that fixed in a W-something?") into a checked, versioned assertion. It is exactly the class of value IBM support historically supplied only reactively, made continuous.

10.4. Justified recommendations: the formula, the rule, and the source

A recommendation the operator cannot verify is a recommendation they cannot safely apply. Every proposed configuration value is therefore shown with the **arithmetic that produced it, the verbatim IBM rule it follows, and a link to the source page** — not a bare number. When the physical- and logical-log sizing advisor proposes a size it states the formula, quotes the IBM guideline (for example `"Set LTXEHWM 30 percent larger than LTXHWM"`), translates the percentage watermarks into GB, and links the parameter's own documentation. Percentages become tangible sizes, and every number is checkable by hand.

The same discipline produces a **configuration-posture view** of a whole subsystem: the log, checkpoint and long-transaction parameters are scored together — each OK, review or action — so the operator sees at a glance whether the settings that can freeze the entire server are safe: a physical log that forces blocking checkpoints, a long-transaction watermark that suspends every thread, or dynamic logging switched off, each with its live value beside its recommended one.

Underpinning both is a **bundled reference library**. The console documentation portal and the MCP resource set ship topic guides — such as `Log, checkpoint and transaction sizing` — that explain the model behind a finding, cross-linked to the live per-server verdict. The expertise is not only encoded in the rules; it is written down, cited to IBM's own documentation, and readable by an operator or an LLM.

10.5. The symptom-first guide

For the operator who starts from a complaint rather than a metric, a `SymptomGuide` offers 14 symptom cards ("CPU-bound / slow response", "contention / blocking", "replication / HA health", "running out of space") that map each symptom to the categories and probes that diagnose it, colouring each probe by the severity of any finding attached to it. It is the same symptom-to-cause method that manual, after-the-fact diagnosis so often lacks.

10.6. The self-check: a credibility gate on the diagnosis

A triage draws on many independent surfaces: the probe grids, the synthesis narrative, the architect verdicts, the trace-profile card, the findings. Each reads the collected data slightly differently, so they can drift out of agreement. A `ConsistencyAudit` runs over the finished report and encodes their expected agreements as machine-checked **invariants**, catching contradictions such as:

- a seq-scan plan shown while the sequential-scans grid is empty;
- a `trace_status` of "OFF" while thousands of statements were captured;
- a finding pointing at a probe that was never collected.

Each invariant was distilled from a real inconsistency found in the field, so the growing set doubles as a regression guard. It never changes the data. It is a credibility gate that lets the system **detect its own incoherence instead of presenting it confidently**, classifying each violation as CRITICAL (two surfaces state opposite facts), WARNING (a strong smell) or INFO (a coherence note). For an evidence-led engagement, a diagnosis that has audited itself for self-contradiction is worth more than one that merely sounds authoritative.

The result is not buried in a log. It is **surfaced as a confidence badge at the top of both the console triage page and the rendered report**, so a reader sees at a glance whether the diagnosis is internally coherent. And it carries through into continuous monitoring: when a watch cycle raises an alert from a triage that failed its own self-check, that alert is **tagged LOW-CONFIDENCE rather than suppressed**. A real critical is never hidden, but the recipient is told, up front, that the underlying numbers may be unreliable and warrant a second look. The platform would rather admit doubt than assert a number it cannot stand behind.

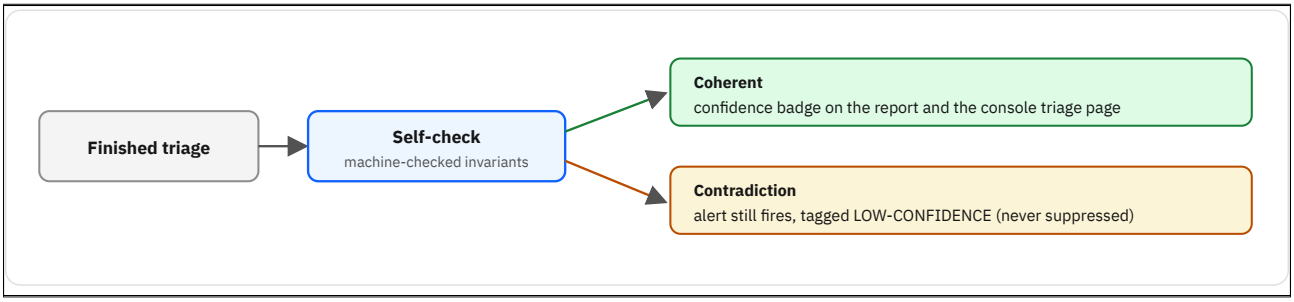

Figure 8 — The self-check as a credibility gate: a coherent triage is badged confident, an incoherent one still alerts but is tagged low-confidence

11. From Data to Decision: the Pipeline and the Deliverable

The pipeline is a single, testable line from a connection to a boardroom-grade document.

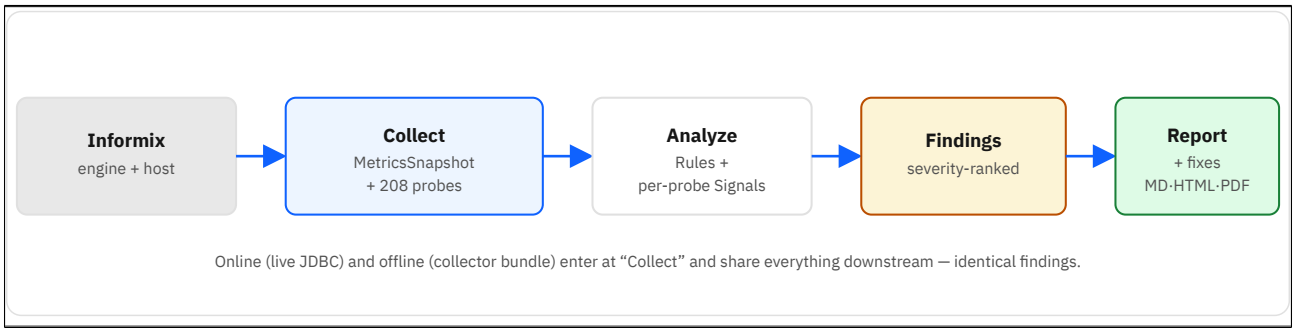

Figure 9 — Diagnostic pipeline: engine and host to collect to analyze to findings to report

11.1. Sampling the worst moment

A single snapshot averages a spike away. `Triage.runSampled(N, interval)` takes N snapshots over a window and diagnoses the **worst observed moment** (the sample scoring the most and severest findings), while also computing per-second interval rates. Crucially, the offline collector samples the same way and the importer applies the `same` worst-moment selection, so a spike caught by a cron job on the host is diagnosed identically to one caught live.

11.2. The professional report

One XML document (in this very documentation grammar) is transformed through two stylesheets into a self-contained, Deister-branded deliverable. Because HTML and PDF share that one source, they cannot drift. Its sections follow an enterprise narrative order:

- **Executive abstract.** States plainly that this is a read-only triage, with the severity counts up front.
- **System under analysis.** Identity of the instance, including the machine UUID and its HA role.
- **Collection parameters.** Trace state, probe pass/na/error counts, sampling and the per-probe timeout. The methodology is disclosed, not hidden.
- **System magnitude.** A panel of sourced, hand-checkable sizing figures.
- **High-availability topology.** The cluster diagram.
- **Overall posture.** The wait-time breakdown and the architect's narrative.
- **Key metrics**, then **the findings**, each a coloured callout carrying its evidence, its "why it matters" and its "recommended action".
- **Diagnostic methodology** and an **evidence appendix.** The methodology states the posture in one line: `single JDBC connection, read-only, no SSH or root`.

A **condensed "action report"** variant strips everything but the problems and their fixes, which is what an operator wants at 3 a.m., while the full variant remains the auditable record.

Real console captures of a triage result and a single finding appear in the console appendix.

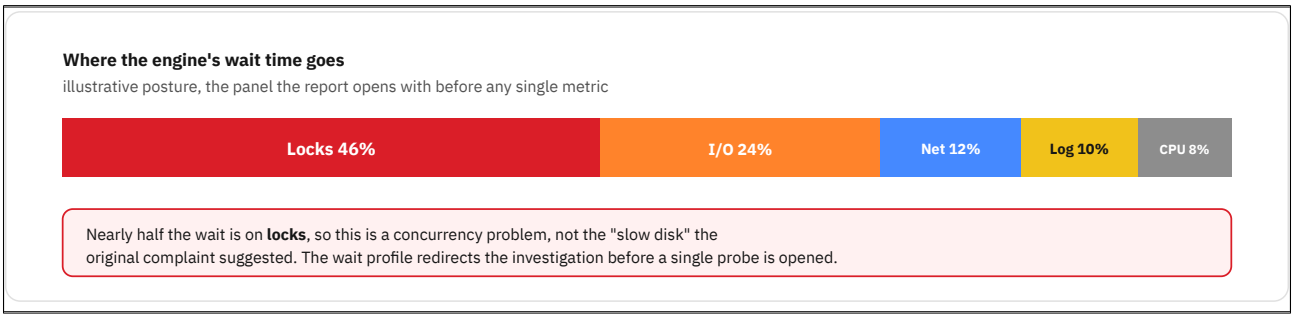

Figure 10 — Where the engine's wait time goes: the overall-posture wait profile that opens the report

The exported report is no longer a lesser view of what the console shows: the HTML deliverable now renders the **same coloured, sortable evidence grids as the live console** (including a dark Carbon theme), so a finding studied in the browser and the same finding in the shared PDF are visibly the one artifact. In the statement-forensics grids each statement carries its **executed plan inline** (a `› plan` disclosure), rebuilt from the trace iterator with estimated-vs-actual rows per operator and a back-link from the plan to the statement that produced it, the "which statement, and what plan did it actually run" question answered in one place rather than across three tabs.

11.3. Plain data is portable data: the single-source bundle

Every run is persisted as a bundle whose `collected_data/` is the one canonical format:

- `metrics.properties` (the scalar metrics);
- a handful of TSV files (dbspaces, sbspaces, users, statements, plan iterators, config);
- the bounded `online.log` tail;
- `probes/<id>.txt`, where each probe's output is canonical TSV.

The **same bundle** is produced live over JDBC and offline by the host collector, and one renderer turns it into the same coloured, sortable evidence tables everywhere: live drill-down, stored session, or uploaded file. A stored bundle can be **re-analysed with today's rules without reconnecting**, and a "known-good" baseline can be pinned so the console reports `what changed` against it.

12. Operating Models: Delivering This as an Enterprise Service

The same core supports a spectrum of engagements, from a one-shot health assessment to a managed, always-on service across a fleet. Critically, it accommodates the security realities of large enterprises where a central tool `cannot` be granted JDBC to every production box.

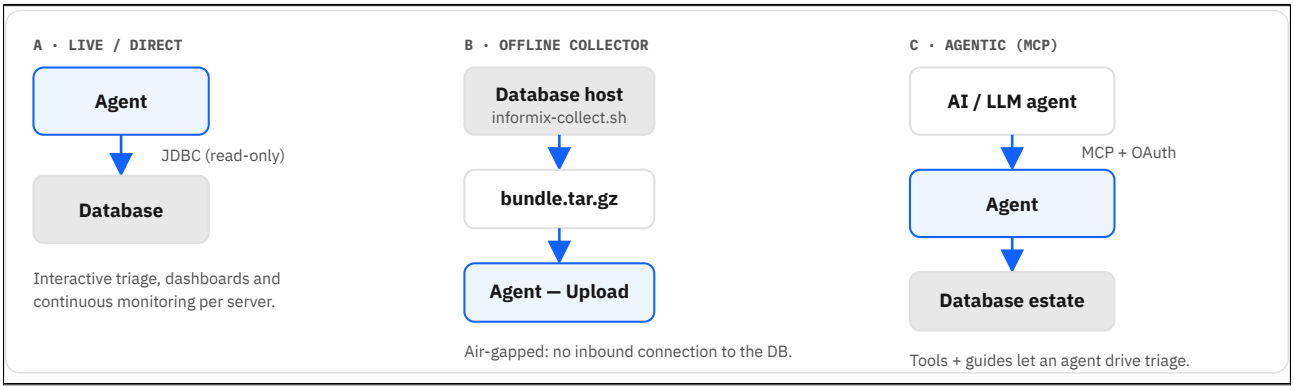

Figure 11 — Three deployment topologies: live direct, offline collector, and agentic MCP

Table 28 — Deployment topologies

Model	How data flows	Fits
Direct (live)	Console/agent connects over JDBC, triages on demand or on a schedule.	Servers the central agent may reach directly.
Offline collector	A generated shell script runs on the host as `informix` (dbaccess/onstat/ proc), emits a bundle.	Air-gapped or JDBC-restricted servers; the collector needs no inbound access.
Spool / MFT drop	Any `.tar.gz` landing in a watched spool directory is routed by its machine UUID and imported.	SFTP / managed-file-transfer estates.
Mail ingestion	The collector e-mails its own bundle; an IMAP poller drops it into the spool.	Sites where mail is the only egress a cron host has.
Continuous monitor	Periodic triage, new-finding diff, alert only on `new` findings.	Managed-service, fleet-wide watch.

Every path converges on **one ingestion pipeline**. A bundle is routed to its server by an embedded, mandatory machine UUID (the importer rejects an anonymous or mismatched bundle), analysed with the current rules, diffed against the server's previous session, and any new findings are pushed to the alert stack. Alert channels are environment-configured and fan-out-isolated: an SMTP mailer and a neutral-JSON webhook (shaped downstream into Slack, Teams or a messaging gateway) fire at or above a configurable severity, and a channel failure is logged, never allowed to break the cycle that raised it.

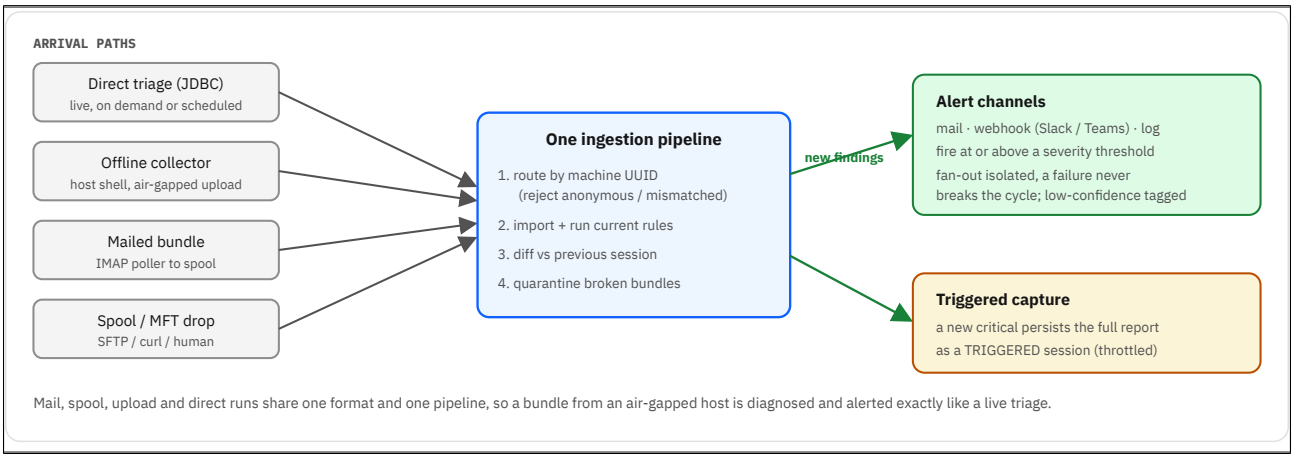

Figure 12 — Every arrival path converges on one ingestion pipeline, which diffs against the prior session and fans new findings out to the alert channels

The capability that closes the central gap

Triggered capture. When a watch cycle sees a `new` critical finding, the agent persists the entire report and evidence set as a `TRIGGERED` session, throttled to one per server per 30 minutes. This is precisely the artifact that after-the-fact diagnosis chronically lacks: **evidence captured automatically at the instant the problem first appears**, not hours later when a human logs in to find the SQL-trace ring long since wrapped. It converts "collect when it recurs" from a manual, usually-failed instruction into an automatic guarantee.

13. Agentic Operation: the MCP Surface

The whole diagnostic capability is exposed over the **Model Context Protocol**, so an LLM agent can operate the platform conversationally. This is not a novelty; on a fleet it is a force multiplier, because it lets a specialist (or an automation) ask "`what is wrong with server X, and why`" and have the model orchestrate the exact probes and plans a human would, then reason over structured results.

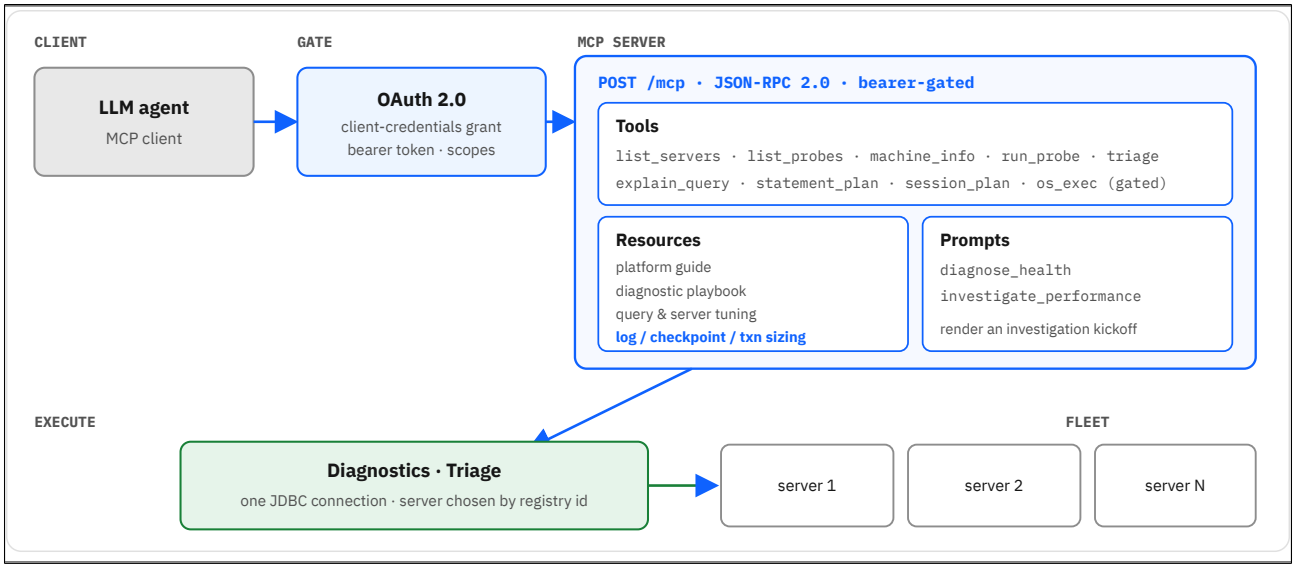

Figure 13 — The agentic surface: an LLM operates the whole platform over MCP, gated by OAuth, across a fleet

Table 29 — MCP tool surface (mcp.Tools)

Tool	Function
`list_servers`	The configured servers (id / label / host / port / database).
`list_probes`	The probe catalog, optionally filtered by category.
`machine_info`	`sysmaster:sysmachineinfo` for a server.
`run_probe`	Run one probe by id and return its output.
`explain_query`	The optimizer plan via `ifx_explain`, **without executing** the query.
`statement_plan`	The executed plan tree of a traced statement (est vs actual per operator; misestimates flagged).
`session_plan`	A live session's open cursors and running plan operators, read-only, no trace needed.
`triage`	A full triage as JSON: posture, where time goes, primary suspect, themes, differential, findings.
`os_exec` (gated)	A host command as `informix`, **off by default, not advertised** unless explicitly enabled.

The surface ships with **resources** and **prompts** that teach a model how to use the tools and how to investigate, so the encoded expertise travels with the API:

- **Resources:** a platform guide, a symptom-to-category diagnostic playbook, query- and server-tuning guides, and a `Log, checkpoint and transaction sizing` reference that explains the subsystem behind those findings.
- **Prompts:** `diagnose_health` and `investigate_performance`.

Two transports are supported: an HTTP JSON-RPC endpoint **gated by an OAuth 2.1 client-credentials grant** (opaque bearer tokens, per-client scopes that filter the tool list, secrets stored only as salted hashes), and a newline-delimited **stdio** transport for a client that owns the process. The host-exec tool is gated independently of everything else, and the console warns when it is on.

Why 'explain without executing' matters here

`explain_query` returns the optimizer's plan for a statement `without running it`. On a system already under stress, that is the difference between diagnosing a bad plan and making the situation worse. It is the same philosophy as the whole platform, observe and never perturb, expressed at the level a language model operates.

14. Security and Governance Posture

An instrument granted access to production databases is itself part of the attack surface, and it is judged accordingly.

Table 30 — Governance controls

Control	Mechanism
Least privilege	Read-only inspection as an unprivileged `informix` user; no root, no SSH.
Minimal footprint	One optional UDR, dropped on `uninstall()`; per-session temp files removed immediately.
Credential secrecy	Server passwords encrypted at rest with AES-256-GCM; plaintext lives only in memory; key from a passphrase (PBKDF2) or an auto-generated 0600 key file.
Self-identification	Every session and statement labelled, so the agent's own traffic is auditable and excluded.
Gated host exec	The `os_exec` tool is off by default and not advertised; the console warns when enabled.
Authenticated API	MCP over HTTP requires an OAuth bearer with per-tool scopes; the browser console is login-gated with an HMAC-signed cookie.
Transport integrity	Bundle import is path-traversal-safe and machine-UUID-gated; unroutable bundles are quarantined with a reason, never silently dropped.

15. The Commercial Case: a High-Assurance Managed Service

The technical properties above underwrite a specific commercial proposition: a **high-assurance, assisted-monitoring service for large and complex Informix estates**, priced and delivered as an enterprise engagement. Its value follows directly from the failure modes set out at the start of this document. Each chronic operational pain is met by a specific platform mechanism with a measurable business outcome.

Table 31 — Value thesis: pain, mechanism, outcome

Chronic pain in large Informix operations	Platform mechanism	Business outcome
Incidents that close without a confirmed root cause because the evidence had evaporated.	Deterministic rules + causal synthesis + triggered capture.	Root cause established on first occurrence; fewer repeat incidents.
Diagnostic data requested after the fact and rarely available in time.	Automatic evidence capture at the moment of detection.	Mean-time-to-diagnosis collapses; support cycles shorten.
Standing risks (a filling lock table, an end-of-support engine, over-provisioned VPs) that are visible in the data but unwatched.	Continuous triage across 18 engine-and-host categories with severity alerting.	Latent risks surfaced before they become outages.
A known, already-fixed engine defect re-encountered after a migration.	Version-to-APAR knowledge base, matched continuously.	Defect exposure known before deployment, not after an ABEND.
A replication layer running without continuous visibility.	First-class HDR/RSS/SDS + ER + Connection Manager probes and rules.	Lag, backlog and divergence caught while still recoverable.
AIX estates underserved by generic tooling.	Per-platform probe specs and a POWER-aware sizing layer.	Parity of insight across a mixed Linux/AIX fleet.

15.1. Engagement shapes

- **Point-in-time architecture & health assessment.** A bounded engagement producing the branded PDF (posture, magnitude, findings, prioritised remediation) for an estate or a flagship instance. Deliverable-led, low-friction (offline collector where JDBC is not granted).
- **Continuous managed monitoring.** Fleet-wide periodic triage with new-finding alerting, triggered capture, and a rolling history; sold per instance or per estate, optionally fronted by the MCP surface for an AI-assisted operations tier.
- **Embedded advisory.** The platform as the standing evidence layer under a retained senior-DBA relationship: the human reasons; the agent guarantees the evidence and does the continuous watching that no human can sustain across thirty instances.

The one-line positioning

A diagnostic model that reasons like a senior Informix architect, encoded, continuous, auditable, and non-intrusive, with the evidence always already captured. It does not replace the specialist; it removes the constraint that has historically defeated the specialist, which is never having the data at the moment it mattered. And because that model speaks the same protocol as modern AI agents, it plugs straight into an AI-assisted operations practice: a reasoning engine your team can audit today and your automation can call tomorrow.

16. Roadmap and Boundaries

The platform is deliberately honest about its edges; a document for specialists is worth more for what it declines to overclaim.

- **Replication depth.** The per-mode captures (`onstat -g dri / rss / sds / smx / proxy / rqm / cmsm`) are now standalone probes; what remains planned is richer ER apply-thread introspection. The current coverage already watches lag, queue backlog, peer state, encryption and clock skew continuously.
- **Rolling trend intelligence.** A rolling multi-interval history with trend comparison, extending the present per-server series and the pinned-baseline "what changed" so a slow drift (a cache ratio decaying over weeks, a lock table filling over days) is caught as a trajectory, not just a threshold breach.
- **OS-agent depth.** Swap-activity rates from `/proc/vmstat` , and correlation of the hottest Informix threads to their host OS threads, closing the last gap between an engine-level symptom and the operating-system thread actually consuming the core.
- **Message-log and stuck-transaction intelligence.** Deepen the `online.log` event model and the blocked-state / long-transaction (XA) detection, so a checkpoint hang, a dynamic-log emergency or a stuck distributed transaction is not only detected but narrated with its precedent events.
- **An ifxcollect archive ingester.** A viewer that maps an existing `ifxcollect run.<timestamp>/` directory onto the agent's onstat-probe evidence blocks (`onstat.g.ckp` to checkpoints, `onstat.g.cluster` to HA, `onstat.g.seg` to memory segments, and so on), so a site that already produces ifxcollect archives for IBM can review them in the console with the per-probe guidance overlaid. Declared scope: a `viewer` , because the rules and architect opinions need the structured SQL / `sysmaster` / `/proc` metrics an ifxcollect archive does not contain.
- **Finer agentic authorization.** Per-tool OAuth scopes on the MCP surface, so an automation can be granted read-only triage without the ability to run a probe or a host command.
- **Console depth.** Live host tiles (free / df / top) and time-series charts alongside the existing dashboards, plus a rendered plan-tree graphic for executed-plan analysis.
- **Portability guarantee (a boundary, not a task).** The probe catalog is verified across Informix 12.10 / 14.10 / 15, with declared, loud handling of version-specific object gaps, so a future probe's typo still fails visibly rather than silently degrading to "not applicable".

> Everything described in this document is implemented and test-backed against live engines: the online and offline paths are proven byte-for-byte equivalent across all three supported Informix major versions by a full-path test that runs the generated collector inside a real container and reconciles it against a live JDBC triage. This is shipped behaviour as at the date of publication rather than a roadmap — and, as set out under `License, Trademarks and Disclaimer` , it is a description of the product, not a warranty or a commitment.

17. Appendix: The Agent Console in Practice

The captures below show the capabilities described throughout this document as they appear in the agent console. They are collected here, rather than inline, so the body reads as one continuous argument; each corresponds to a section above.

Every capture comes from one triage session against a production Informix 14.10 primary — a single run, 214 probes, 4,602 traced statements. Instance, host, database, schema and account names have been replaced throughout; all measurements, timings and verdicts are exactly as the agent produced them.

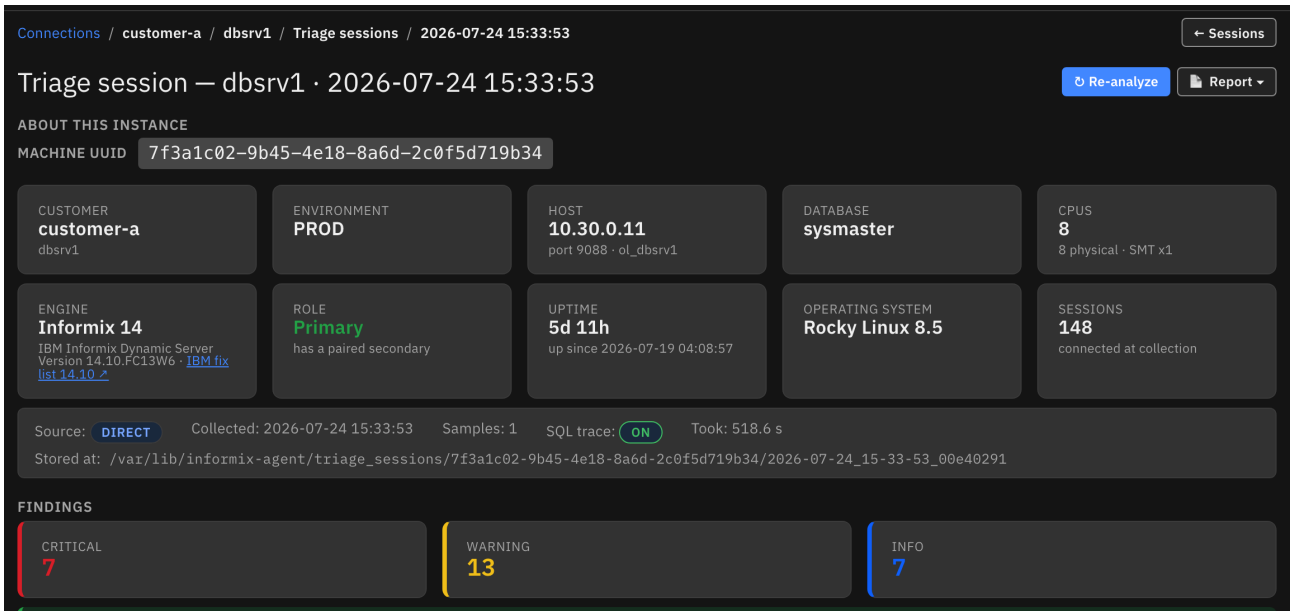

Figure 14 — A triage session: the instance under analysis, how the collection was taken, and the severity counts it produced

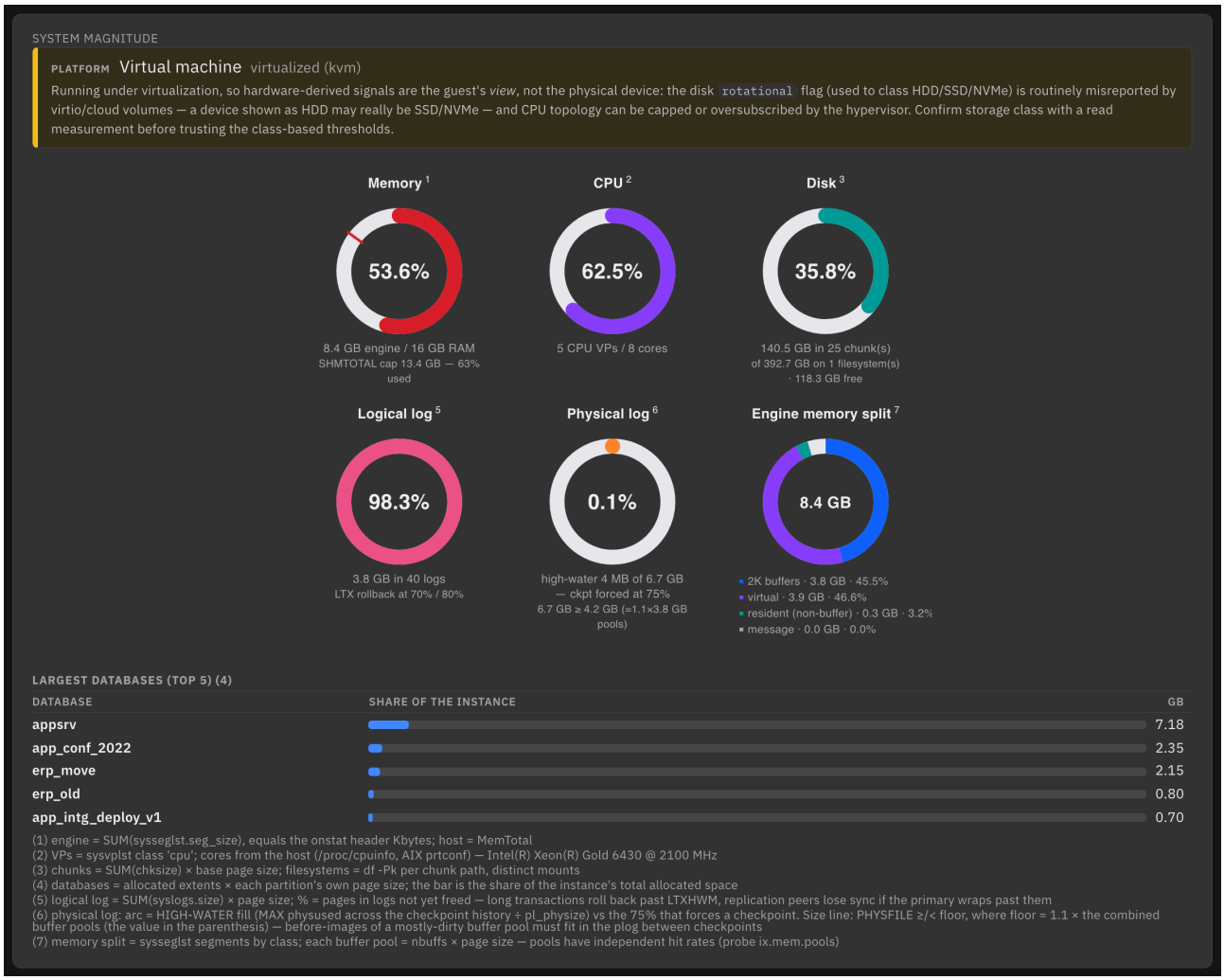

Figure 15 — System magnitude: memory, CPU, disk and log capacity read from the engine, each with the derivation that produced it

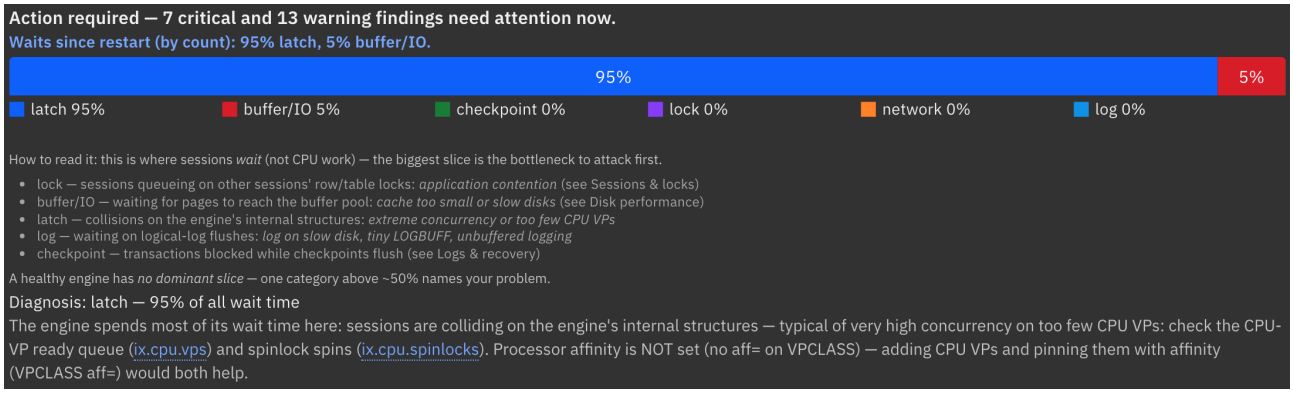

Figure 16 — Where the engine waits: the wait-time split, and the diagnosis the dominant category implies

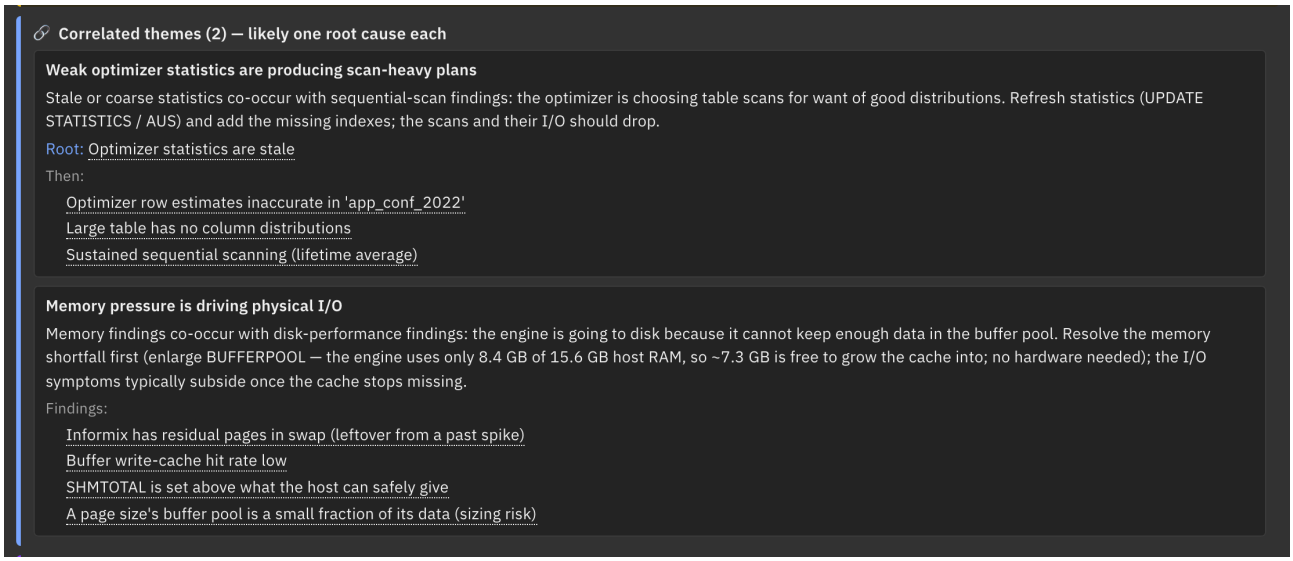

Figure 17 — Causal synthesis: findings grouped into themes, each with the root to fix and the downstream findings expected to subside with it

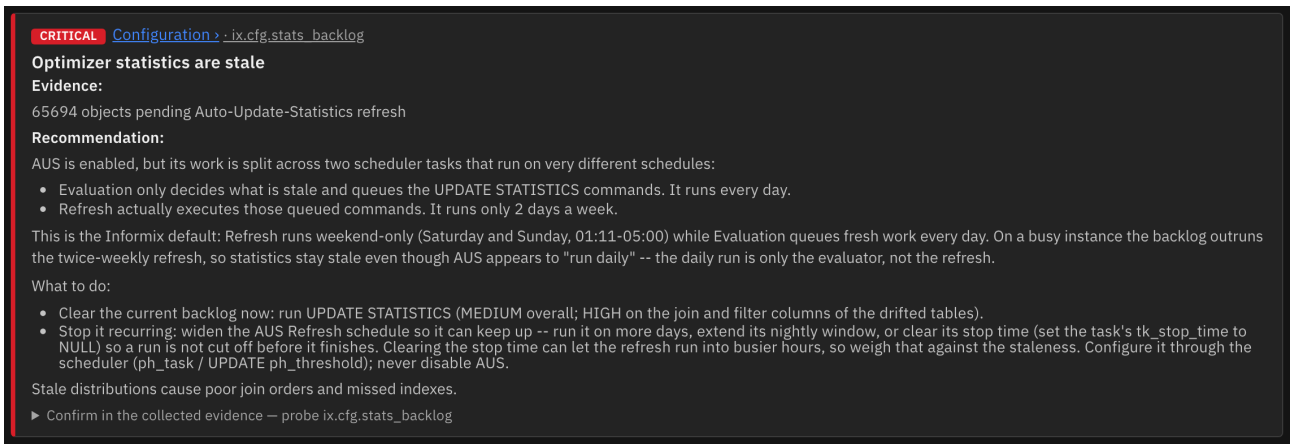

Figure 18 — A single finding as the reader sees it: severity, evidence, the recommended action and the probe that will confirm it

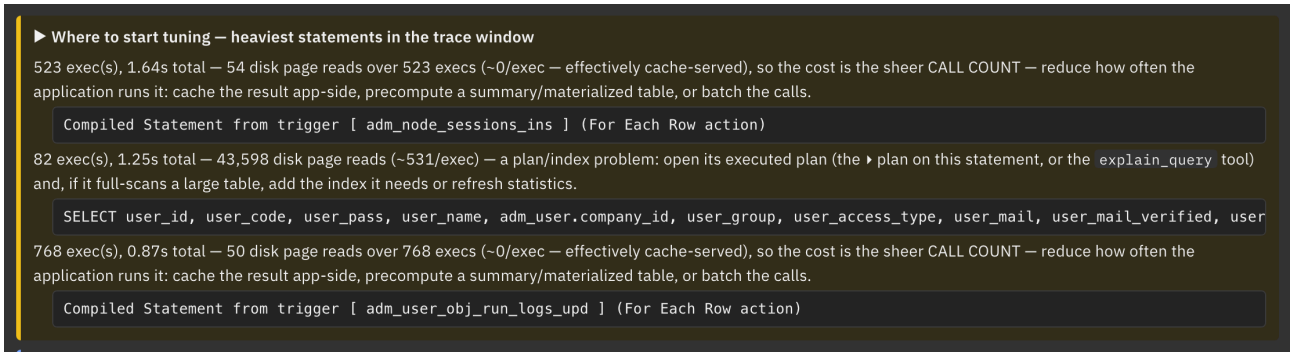

Figure 19 — Where to start tuning: the heaviest statements of the trace window, each with the reason it is expensive

Figure 20 — Bounded capture: the agent sizes the SQL-trace ring against the engine's own shared memory, and shows every step of the derivation

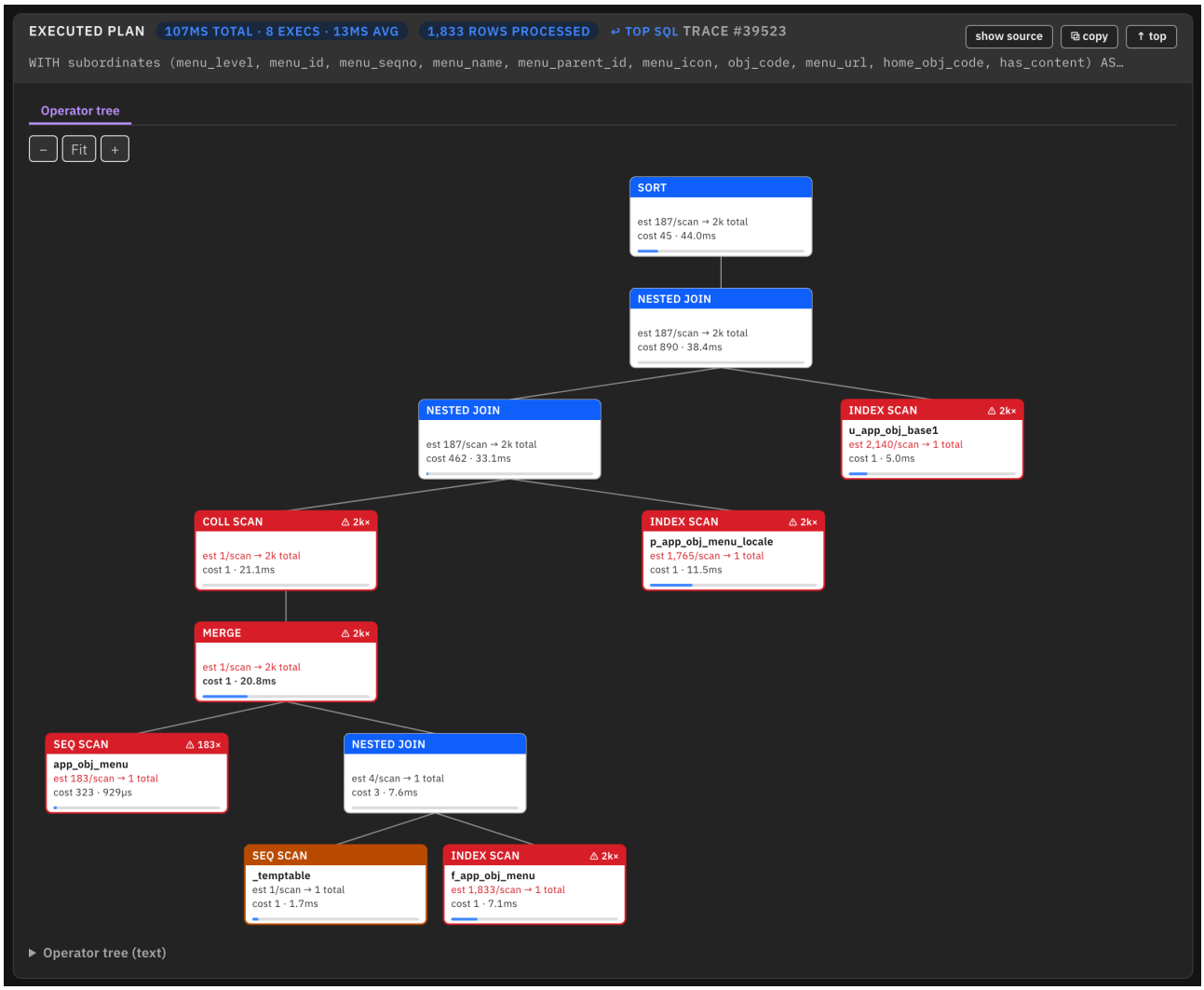

Figure 21 — The executed plan of a traced statement: estimated versus actual rows per operator, with cardinality misses flagged in red

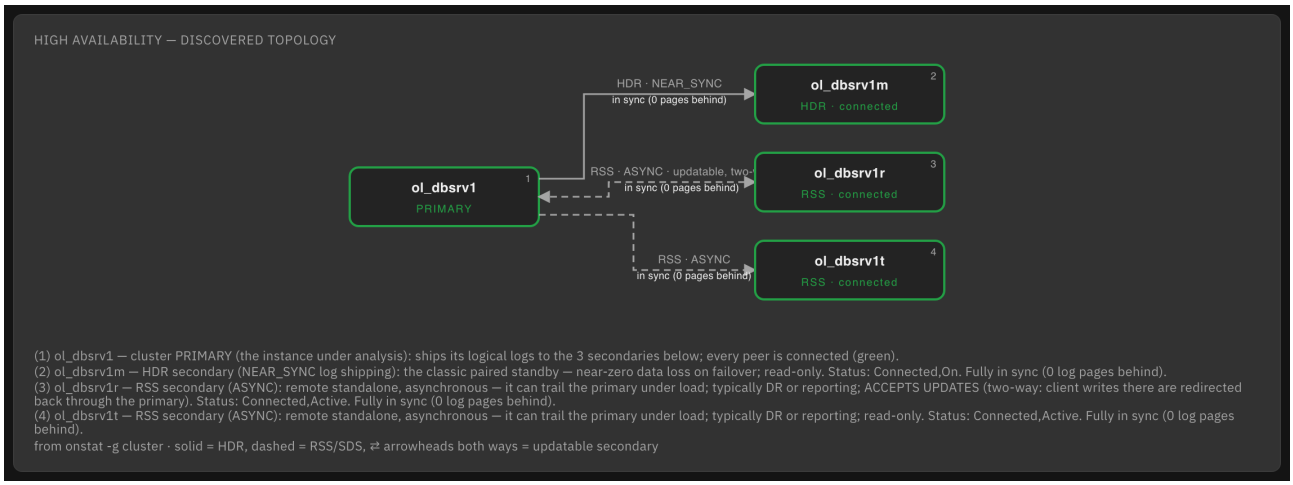

Figure 22 — Replication and high availability: the cluster topology discovered from the primary, with the log-shipping mode and lag of every peer

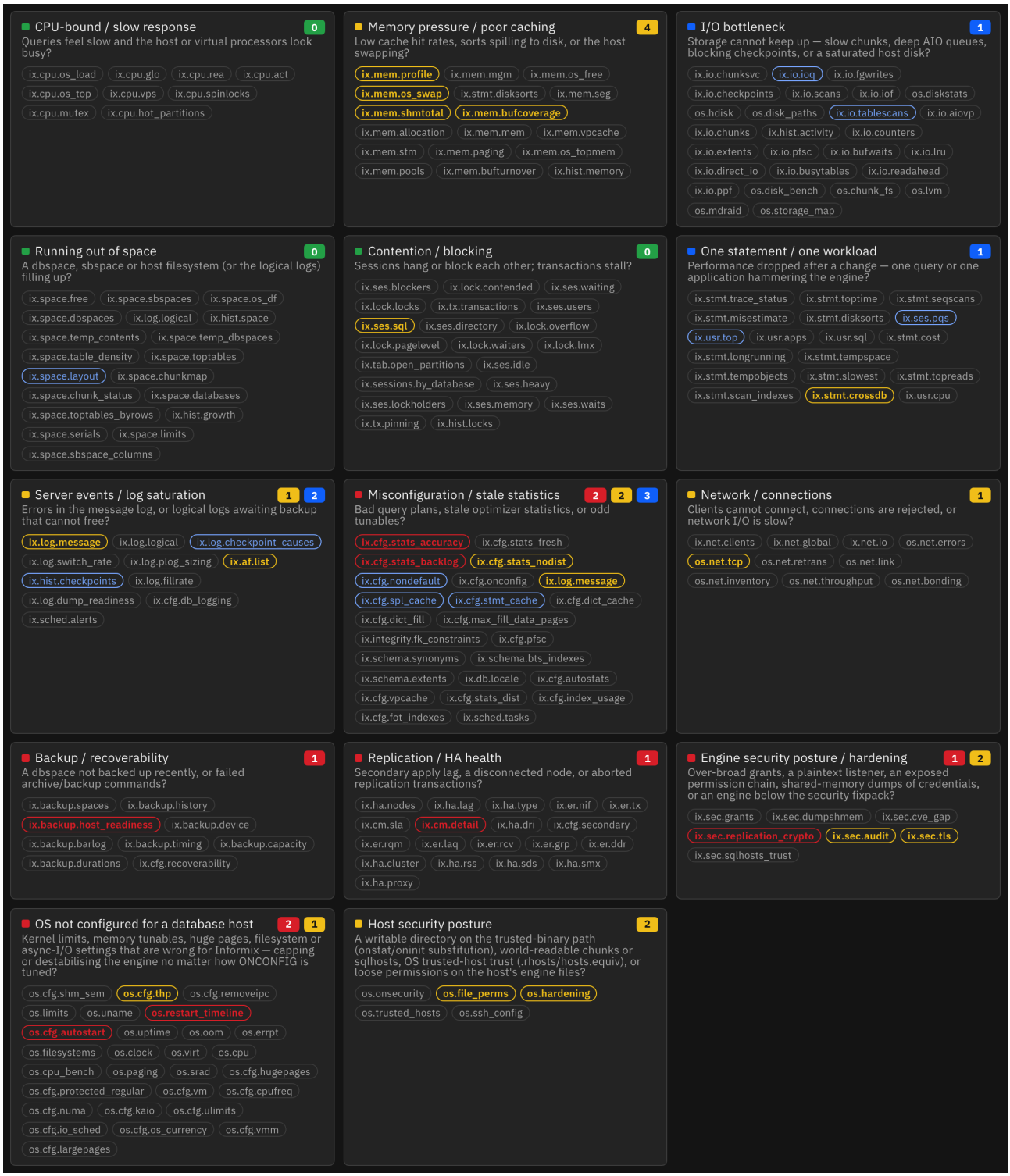

Figure 23 — The symptom-first guide: start from what the operator is seeing, and the probes for that area open with their current status

18. Glossary — Diagnostic and Informix Terms

A reference for the vocabulary used throughout this document — the agent's own concepts (probe, signal, rule, finding, triage) and the Informix engine and host-platform terms they rest on.

Table 32 — Diagnostic and Informix glossary

Term	Definition
Probe	A single read-only measurement, classified by category and sourced from `onstat`, a `sysmaster` query, or a host command. The catalog holds 208 of them.
Category	One of the 19 diagnostic areas a probe belongs to (Configuration, CPU, Memory, Logs, Disk, Sessions & Locks, SQL profiling, OS config, Security, …).
Signal	A per-probe column threshold that marks an individual value as offending, so a grid highlights the exact cell that is out of range.
Rule	A deterministic check that reads structured metrics (never parsed `onstat` text) and emits findings; the engine ships dozens of them.
Finding	A rule's conclusion — a severity (info / warning / critical), the evidence behind it, why it matters, and the recommended action.
Triage	A full run: collect metrics and probe evidence, apply the rules, then rank the findings into a single most-likely root cause.
Primary suspect / causal rank	The ranking that orders symptoms by how far `upstream` they sit in the causal chain, so a consequence is not mistaken for a cause.
Self-check	A credibility gate: a triage whose findings form a coherent causal story is badged `confident`; contradictory evidence is flagged instead of asserted.
Architect opinion / verdict	A deterministic, templated conclusion attached to a probe block that reads the metrics and states, in plain terms, whether the area is healthy and why.
Snapshot vs rate	A snapshot is a point-in-time metric set; a rate is the per-second delta between two snapshots over an interval, so bursty activity is visible.
Session	A persisted triage run on disk — metadata, the collected data, the findings, and the rendered report — that can be re-analysed later without reconnecting.
Bundle / collector	A self-contained `tar.gz` of the canonical `collected_data`, produced on the host by the offline collector script and ingested by the same pipeline as a live run.

Term	Definition
onstat	Informix's shared-memory monitoring utility; the agent captures its output over SQL through the Admin API rather than over a shell.
sysmaster	The pseudo-database of in-memory system views (sessions, locks, checkpoints, the SQL-trace ring, chunk usage) that the engine exposes to SQL.
SQL Admin API	The `sysadmin:task()` / `admin()` functions that run `onstat` and administrative commands from within SQL over the one JDBC connection.
Checkpoint	The periodic flush of modified buffers to disk that establishes a recovery point; `Blocked:CKPT` is the state where sessions stall while one completes.
Physical log (PHYSFILE)	The before-image area used to make checkpoints and fast recovery consistent; it is recycled at each checkpoint.
Logical log (LOGSIZE / LOGFILES)	The ring of transaction-log files; the sizing target is to switch a log no more often than once every 15 minutes.
LTXHWM / LTXEHWM	Long-transaction watermarks (percent of total log): the high-water mark that starts rolling back the offending transaction, and the exclusive-access mark — the “emergency brake” — that freezes other work to save the log.
Dbspace / chunk	A logical storage container and the physical file (or device) that backs it; a dbspace is made of one or more chunks.
Sbspace	A storage space for smart large objects (BLOB/CLOB and the internal LO handles used by, for example, `FILETOCLOB`).
PDQ	Parallel Database Query — the engine's intra-query parallelism, gated by memory grants; a starved grant queue serialises heavy queries.
Sequential scan / light scan	A full read of a table's rows; a `light scan` reads around the buffer pool for large scans. Heavy sequential scanning is a classic sign of a missing index or stale statistics.
SQL-trace ring	`syssqltrace`, an in-memory ring buffer of completed statements with their cost columns; the agent ranks it by cumulative execution time to surface the costliest statements.
Statement fingerprint	A statement with its literals stripped, so many executions that differ only by their constants group into one logical statement for profiling.
Executed plan / plan iterator	The per-operator actuals of a traced statement (from `syssqltrace_iter`) — estimated versus actual rows per node — used to spot cardinality misses and plan flips.

Term	Definition
Buffer cache read / write %	The read and write cache hit ratios derived from `sysprofile` counters; a falling read cache % is an early memory-pressure signal.
Update statistics / AUS	The maintenance that refreshes optimizer statistics; Auto Update Statistics (AUS) is the built-in scheduler, and a growing backlog means the optimizer is planning on stale data.
Host agent (remote exec)	The mechanism that runs read-only host commands (`ps` , `vmstat` , `df` , `/proc`) as the unprivileged `informix` OS user over the same JDBC connection — no SSH or root.
THP	Transparent Huge Pages — a Linux memory feature that is generally harmful for database workloads and is checked by the OS-config probes.
LPAR	Logical Partition — the IBM POWER/AIX virtualization unit; the agent reads its entitlement and shared-vs-dedicated mode to judge whether the engine has the CPU it assumes.
MCP	Model Context Protocol — the JSON-RPC surface (tools, resources, prompts) over which an LLM agent operates the whole platform conversationally.
OAuth client-credentials	The machine-to-machine grant that gates the MCP surface: a client exchanges an id and secret for a short-lived bearer token.

19. License, Trademarks and Disclaimer

The Informix Diagnostic Agent is a **proprietary, commercial software product** owned and published by **Airtool Technologies** (airtool.io). **All rights reserved.** It is licensed, not sold, and its use is governed by a commercial license agreement. No right to use, copy, modify, distribute, sublicense, or create derivative works is granted except as expressly permitted in a written license from Airtool Technologies. Unauthorized reproduction or distribution is prohibited. Contact info@airtool.io for licensing terms.

Purpose of this document. This document is provided for information only. It describes the product as at the date of publication, is subject to change without notice, forms no part of any agreement, and creates no representation, warranty or commitment. The terms governing any use of the software are those of the written license agreement between you and Airtool Technologies.

Warranty. Warranties, if any, are those expressly set out in that agreement. Except as expressly so provided, and to the maximum extent permitted by applicable law, the software is provided **"as is"**, and Airtool Technologies disclaims all other warranties and conditions, whether express, implied or statutory, including the implied warranties of merchantability, fitness for a particular purpose and non-infringement.

Limitation of liability. Except as expressly provided in that agreement, and to the maximum extent permitted by applicable law: neither party is liable for indirect, incidental, special, punitive or consequential damages, or for loss of profits, revenue, data or business, however caused; and each party's total aggregate liability is limited as set out in that agreement. Nothing in this document excludes or limits liability for death or personal injury caused by negligence, for damage to tangible property caused by negligence, for fraud or fraudulent misrepresentation, for wilful misconduct, or for any other liability that cannot be excluded by law.

Trademarks — no affiliation. IBM®, **Informix®** and **AIX®** are trademarks or registered trademarks of International Business Machines Corporation; Informix is currently developed and distributed by HCL under license. This product is **independent** and is **not** affiliated with, sponsored by, or endorsed by IBM or HCL. Product names are used solely for identification and interoperability (nominative fair use); no third-party logos are distributed.

Commercial support. Licensing, production support (with SLAs), custom development and consulting are available from **Airtool Technologies** — airtool.io · info@airtool.io. Support entitlements are defined by your commercial license or support agreement. Bundled third-party components retain their own licenses.