Empirical Performance Analysis Report

Acumen FHIR Query Translation Engine

Load-Test and Server-Instrumentation Evidence Across Two Hardware Tiers

Classification: Confidential

Reddy and Reddy Family Office (i.f.)
Registered Office: GIFT City, IFSC, Gujarat

July 19, 2026

Abstract

This report presents empirical performance measurements of the Acumen FHIR Query Translation Engine on two hardware tiers—a consumer laptop and a cloud virtual machine—using three mutually independent instruments: k6 HTTP load testing for latency and Service-Level-Agreement (SLA) evidence; external sampling of the server process for CPU attribution; and an in-process telemetry collector inside the server for garbage-collection (GC), event-loop, scheduling, and I/O behaviour, which are observable only from within the engine process. Every quantitative claim is traceable to a cited benchmark JSON file. The principal findings are: (1) **indexed query latency is low and predictable**—p95 of 28–73 ms on the laptop and 24–83 ms on the virtual machine's 83.6× larger corpus, with tail ratios (P99/P50) of 1.1–1.3×; (2) **the engine is single-thread-bound with a per-core throughput ceiling** (≈135–147 req/s VM, ≈57–63 req/s laptop): a controlled 1-to-50 concurrent-user sweep shows per-query server CPU essentially flat in concurrency, latency growth under load being queueing delay measured directly at the server's event loop; (3) **in-process GC telemetry shows the engine's allocation footprint per query is fixed**—GC pause per query is ≈1 ms at every concurrency level on both tiers—and reveals a **discrete, persistent heap-management state transition on the VM** that steps per-query CPU between ≈12 ms and ≈17 ms without materially affecting client-visible latency; (4) **the full write surface—create, conditional update/delete, transaction bundles, and history—executes correctly with cryptographically signed, hash-chained audit entries**; (5) **logarithmic data scaling is confirmed**: an 83.6× larger resident dataset raises the worst indexed-search p95 by only 2.4× and *reduces* the others; and (6) a dedicated **Safety-Kernel absorption benchmark** rejects 100% of attack corpora totalling 50,302 requests across the two machines with zero false positives—with the asymmetries between the two tiers under sustained attack reported honestly. Known SLA breaches (terminology-expansion cascades, two date-prefix operators, and cold-cache reverse chains under one client pattern) are reported alongside the passing results as optimization targets.

Contents

1 Executive Summary

Benchmarks were conducted on **July 18, 2026** against build `21e4d9f` on two environments whose resident datasets were counted directly from the databases. All figures below are sourced from the JSON files enumerated in Section 18.

- **SLA compliance:** 145/153 scenario measurements pass on every full serial run—laptop SSD, laptop HDD, and both same-day VM runs. The stable breaches are terminology-expansion cascades and two date-prefix operators (Section 12); the remaining laptop breaches are write-path scenarios that flicker run-to-run under the 5.94 GB host's memory pressure.
- **Indexed latency:** p95 of 34–73 ms (laptop SSD), 28–47 ms (laptop HDD), and 24–83 ms (VM) on the four core search scenarios, despite the VM serving 83.6× more clinical records.
- **Concurrency (Section 4):** a fixed-workload sweep from 1 to 50 concurrent users shows per-query server CPU flat in concurrency on both machines—≈23–27 ms on the laptop across the whole range; ≈12 ms and ≈17 ms on the VM in its two heap states (below). Throughput saturates at ≈57–63 req/s (laptop) and ≈135–147 req/s (VM); beyond the knee, added users buy latency, not throughput (p95 grows 28× and 11× respectively across the sweep). Both sweeps recorded 0% errors.
- **Server-process telemetry (Section 5):** in-process GC instrumentation shows GC work is proportional to query count, not concurrency—≈0.2–0.33 GC events and ≈1 ms of GC pause per query at every level on both tiers—and event-loop delay measured inside the server rises linearly with concurrency (28→304 ms laptop, 22→166 ms VM), identifying queueing, not slower execution, as the latency mechanism under load.
- **Emergent finding — VM heap-state transition:** between 10 and 15 concurrent users the VM server shifted into a high-GC regime—major-GC activity per window rose ≈7×, peak RSS fell 34%, and per-query CPU stepped from ≈12.1 ms to ≈17.2 ms—and *remained there* even when load returned to a single user. Client-visible throughput changed by only −4% because the added collection work ran on background GC threads. Capacity planning on the VM should budget the high-GC state.
- **Write path (Section 6):** create, read-by-id, conditional create/update/delete, transaction bundles (1/5/10 entries), and history endpoints all execute, each producing a cryptographically signed, hash-chained `ResourceHistory` entry (ECDSA P-256 over SHA-256, chained by `previousHash`). VM write p95: 21–113 ms across the entire surface.
- **Data scaling (Section 7):** 83.6× resident-dataset increase (77,211 → 6,458,565 clinical-core documents) raises the worst indexed-search p95 by 2.4× and reduces the token/date scenarios, consistent with $O(\log n)$ index utilization.
- **Safety Kernel (Section 10):** 26,838/26,838 (VM) and 23,464/23,464 (laptop) attack requests rejected, zero false positives on 4,678 valid queries. On the VM, rejection costs 4.9–5.6× less than a valid query and legitimate traffic shows no measurable degradation under a sustained storm; on the laptop the asymmetry inverts (rejections ≈1.15× the valid-query cost) and valid traffic degrades 3.7×—both reported as measured. Seven candidate attack patterns bypass the guard layer and execute as legal-but-expensive queries (one reaching 20 s on the VM); these are named gaps.
- **Staged stress (Section 9):** a 15/25/50-user staged run sustains a constant ≈30% scenario-mix error rate on both machines (VM overall p95 193 ms across 113,624 requests; laptop 824 ms across 35,908).

2 Test Environments and Datasets

2.1 Hardware and Runtime

Component	Laptop (QUASAR)	VM (ubuntu-8gb-fsn1-1)
CPU	AMD Ryzen 7 3750H (8 cores)	AMD EPYC-Genoa (8 vCPU)
RAM	5.94 GB available	15.24 GB
Storage	SSD / HDD	NVMe SSD
OS	Windows 11 (10.0.26200)	Linux (kernel 7.0.0-27)
Node.js / V8	v22.19.0 / 12.4.254.21	v22.23.1 / 12.4.254.21
MongoDB	8.2 (replica set rs0)	8.0 (replica set rs0)
Build	`21e4d9f`	`21e4d9f`

Table 1: Test environment specifications.

2.2 Resident Datasets (Counted from the Live Databases)

Both datasets were counted directly from the live databases with full `countDocuments()` scans on July 18, 2026. These counts, not source-file estimates, anchor the scaling analysis in Section 7.

Collection	Laptop	VM	Ratio
observations	40,801	3,272,967	80.2×
procedures	15,391	1,395,511*	90.7×
diagnosticreports	11,077	944,933*	85.3×
encounters	5,262	485,651*	92.3×
conditions	3,743	332,679*	88.9×
careplans	194	16,502*	85.1×
patients	743	10,322	13.9×
Clinical-core (7 types)	**77,211**	**6,458,565**	**83.6×**
Total objects (all collections)	175,702	6,469,291	—

Table 2: Resident document counts. *Per-type VM figures are the counts recorded when the corpus was last enumerated in full; the clinical-core total and the patient/observation counts are from the July 18 scan. The patient ratio (13.9×) understates dataset scale because it is diluted by benchmark-created bare patients; the clinical-resource ratios (80–92×) carry the comparison.

2.3 Measurement Methodology

Three independent instruments are used, and the report is careful never to mix them:

1. **HTTP-observable metrics** (latency percentiles, throughput, error rate, SLA pass/fail) come from the load harness, which issues requests over the loopback interface and records wall-clock timing per request. These are externally observable and independent of any server-side accounting.

2. **Per-query server CPU** is measured by sampling the *server process itself* from an external observer—`/proc/[pid]/stat` (`utime+stime`) on Linux and `Get-Process TotalProcessorTime` on Windows—across a fixed workload, and dividing the CPU-time delta by the server-side query count from `/_internal/metrics/export`. As a control, the load-generator process is sampled alongside: its CPU cost stays at 0.9–2.2 ms per query, under a tenth of the server's, confirming that CPU attribution to the engine is clean and that the load generator is never the bottleneck.

3. **In-process server resource telemetry** runs inside the server process and is bracketed around each measured window via internal start/stop endpoints. It records V8 garbage-collection events through a `PerformanceObserver` (count, pause duration, and kind: minor /

major / incremental), event-loop delay through `monitorEventLoopDelay`, and context-switch and file-system operation counters through `process.resourceUsage()`. GC and event-loop behaviour are V8-internal and observable only from within the process, which is why this instrument exists. Platform caveat: context-switch counters are available on Linux only (Windows reports zero), and the OS I/O-operation counters differ in semantics—Linux counts file-system blocks written by the process, Windows counts write system calls including socket writes; the report interprets each accordingly.

Server-side query counts are read from `serverMetrics.totalQueries`; optimization activity from `serverMetrics.optimizationCounts`. All raw files are listed in Section 18.

3 Query Latency

3.1 Cross-Environment Comparison

Table 3 reports p95 latency for the four core indexed-search scenarios (100 measured iterations each after 20 warmup iterations). The VM serves 83.6× more data than the laptop; that the p95 figures remain comparable is the scaling result quantified in Section 7.

Scenario (p95)	Laptop SSD	Laptop HDD	VM NVMe
STRING_SEARCH_EXACT	34.13 ms	27.87 ms	83.07 ms
TOKEN_SEARCH_SINGLE	73.10 ms	47.03 ms	31.77 ms
TOKEN_SEARCH_OBSERVATION	44.04 ms	36.62 ms	24.45 ms
DATE_RANGE_SEARCH	53.80 ms	44.87 ms	30.37 ms

Table 3: Indexed-search p95 latency. Sources: `benchmark-2026-07-18T10-20-50-{ssd,hdd}.json` (laptop), `benchmark-2026-07-18T13-58-39.json` (VM). Token- and date-indexed scenarios run *faster* on the VM despite its 83.6× larger corpus, reflecting server-grade hardware; `STRING_SEARCH_EXACT` is the one scan-heavier scenario that pays for corpus size. A same-day repeat VM run (`11-18-10`) reproduces every row to within 3 ms.

3.2 Warmup Transition

Per-scenario warmup windows capture the cold-to-warm transition. On the laptop SSD, `STRING_SEARCH_EXACT` peaks at 112.8 ms during warmup against a 25.5 ms steady-state average (4.4×); on the VM, 115.8 ms against 74.8 ms (1.5×). The starkest case is the reverse-chain family on the VM's production-scale corpus: the warmup window contains a first-iteration cost of 8,503 ms, after which steady state settles at a 45.2 ms average—the signature of a large one-time cache/pipeline build cost that subsequent queries amortize (see Section 12 for where this cold cost resurfaces).

3.3 Tail-Latency Ratios

P99/P50 ratios on indexed scenarios are well below the 5–10× common in database-backed services:

Scenario	p50	p99	P99/P50
STRING_SEARCH_EXACT (VM)	75.8 ms	83.8 ms	1.10×
TOKEN_SEARCH_OBSERVATION (VM)	21.6 ms	26.4 ms	1.23×
DATE_RANGE_SEARCH (VM)	25.5 ms	33.1 ms	1.30×

Table 4: Tail-latency ratios on indexed operations (VM). Bounded query complexity keeps p99 within 10–30% of p50.

4 Concurrency and Per-Query Server CPU

This section reports a controlled concurrency sweep: an identical, valid-only workload (five round-robin search and read-by-id URLs) driven at 1, 2, 5, 10, 15, 25, and 50 concurrent users, with a 20-second warmup and 90-second measured window per level, and a repeat of the first level at the end as a drift check. Per-query CPU is measured against the server process (Section 2.3). Holding the workload fixed while varying only concurrency is what isolates the effect of concurrency from workload composition.

4.1 Results

VU	CPU/query	Throughput	p95	Server util.	Server queries
Laptop (Ryzen 3750H, 77,211-doc corpus):					
1	24.75 ms	49.29 req/s	40.9 ms	97.6%	3,550
2	23.22 ms	63.04 req/s	49.3 ms	117.2%	4,543
5	24.47 ms	60.30 req/s	129.8 ms	118.3%	4,350
10	25.38 ms	57.79 req/s	243.7 ms	117.7%	4,176
15	25.87 ms	58.26 req/s	342.4 ms	121.2%	4,217
25	26.93 ms	56.62 req/s	621.0 ms	123.0%	4,112
50	26.80 ms	57.34 req/s	1152.0 ms	124.8%	4,192
1 (drift)	22.46 ms	53.68 req/s	37.3 ms	96.5%	3,866
VM (EPYC-Genoa, 6,458,565-doc corpus):					
1	13.46 ms	65.13 req/s	51.7 ms	70.2%	4,693
2	11.73 ms	123.09 req/s	54.5 ms	115.6%	8,870
5	11.66 ms	146.83 req/s	76.0 ms	137.2%	10,596
10	12.14 ms	141.46 req/s	122.2 ms	137.9%	10,223
15	17.19 ms	134.73 req/s	178.0 ms	186.4%	9,759
25	17.21 ms	135.88 req/s	286.3 ms	189.2%	9,892
50	17.00 ms	135.88 req/s	574.5 ms	189.1%	10,007
1 (drift)	18.39 ms	63.05 req/s	53.6 ms	92.9%	4,546

Table 5: Fixed-workload concurrency sweep. Sources: `sweep-2026-07-18T10-19-48.json` (laptop) and `sweep-2026-07-18T10-37-57.json` (VM), both under `concurrency-sweep/` (Section 18). Both runs recorded 0% errors at every level. The step in VM CPU/query and utilization between 10 and 15 users is a heap-management state transition analysed in Section 5; note that it persists in the drift row.

4.2 Interpretation: Flat Per-Query Cost, Single-Thread Saturation

Three signatures in Table 5 agree on both machines:

1. **Per-query CPU does not amortize with concurrency.** On the laptop, CPU per query stays within 23.2–26.9 ms across the entire 1–50 range (a 14.7% spread; fitted power-law exponent $|\alpha| \leq 0.055$ at 95% confidence—a total change of at most ≈15% over a 50× concurrency range). On the VM, CPU per query is flat *within each heap state*: 11.7–13.5 ms in the low-GC state (1–10 users) and 17.0–17.2 ms—a 1.2% spread—in the high-GC state (15–50 users); the step between the states is a discrete heap-management transition, not a function of load level (Section 5). **Concurrency neither reduces nor super-linearly inflates the CPU cost of a query.**

2. **Throughput saturates at a per-core ceiling.** Throughput rises only until the server's single JavaScript execution thread is busy—immediately on the laptop (≈57–63 req/s, CPU pinned near 1.2 cores) and after the first doubling on the VM (≈135–147 req/s at 1.4–1.9 cores, the higher figure reflecting background GC threads). Against an ideal 50×, measured scaling is 1.28× (laptop) and 2.25× (VM).

3. **Latency grows by queueing.** Because throughput is capped, additional concurrent users wait: p95 rises 28× on the laptop and 11× on the VM across the sweep, and Section 5 shows the wait is located at the server's event loop.

The engine's per-query CPU cost is therefore a property of the query, the hardware, and the runtime's heap state—not of concurrency. This is a single-thread-bound service: it delivers low latency at low-to-moderate concurrency and degrades by queueing—not by per-query slowdown—once the per-core CPU ceiling is reached. Capacity planning follows directly from the measured ceiling and per-query cost; horizontal scaling (more processes/cores) is the lever for higher throughput, since a single process cannot exceed one core of JavaScript work.

4.3 Energy per Query

Per-query energy follows directly from measured CPU time under a linear power model [Möbius et al., IEEE TPDS 2013; Khan et al., ACM TACO 2018], $P = P_{\text{idle}} + (P_{\max} - P_{\text{idle}})\,U$. Using the measured per-query CPU and an active-power assumption of 8 W:

$$E_{\text{query}}^{\text{VM}} \approx 8\ \text{W} \times (12.1\text{–}17.2) \times 10^{-3}\ \text{s} \approx 97\text{–}138\ \text{mJ}, \qquad E_{\text{query}}^{\text{laptop}} \approx 8\ \text{W} \times 25 \times 10^{-3}\ \text{s} \approx 200\ \text{mJ}.$$

Because per-query CPU is flat in concurrency (Table 5), **per-query energy is likewise flat in concurrency**—though on the VM it inherits the heap-state dependence above (the high-GC state costs ≈40% more energy per query). These figures are modeled from CPU time, not measured with RAPL, and represent the genuine per-query resource cost of the translation-and-safety pipeline—a cost worth reducing, not an efficiency claim.

5 Server-Process Resource Telemetry Under Load

The in-process instrument (Section 2.3, item 3) was bracketed around every measured sweep window, giving a per-concurrency-level view of the engine's internal runtime behaviour: garbage collection, event-loop delay, scheduling, and I/O operations. Table 6 reports the key derived quantities.

VU	GC/query	Pause/query	GC % wall	Major GC	EL delay	Peak RSS	Vol. ctx/q	Invol. ctx
Laptop (win32; context-switch counters unavailable on this platform):								
1	0.227	0.95 ms	3.8%	9	28.4 ms	453 MB	—	—
2	0.223	0.93 ms	4.7%	9	27.1 ms	502 MB	—	—
5	0.210	1.00 ms	4.8%	10	39.2 ms	481 MB	—	—
10	0.203	1.03 ms	4.8%	10	53.8 ms	488 MB	—	—
15	0.210	1.14 ms	5.4%	10	85.0 ms	482 MB	—	—
25	0.210	1.26 ms	5.8%	11	149.7 ms	483 MB	—	—
50	0.213	1.25 ms	5.8%	11	304.2 ms	463 MB	—	—
1 (drift)	0.226	0.92 ms	4.0%	8	27.9 ms	482 MB	—	—
VM (linux):								
1	0.305	0.80 ms	4.2%	17	21.9 ms	616 MB	11.2	304
2	0.304	0.82 ms	8.1%	28	22.8 ms	696 MB	10.2	408
5	0.293	0.86 ms	10.2%	34	25.3 ms	693 MB	7.6	745
10	0.296	0.91 ms	10.3%	34	31.1 ms	691 MB	7.0	679
15	0.324	0.95 ms	10.3%	246	45.9 ms	456 MB	8.2	1,938
25	0.324	0.97 ms	10.7%	239	80.0 ms	456 MB	7.5	1,713
50	0.324	1.00 ms	11.2%	243	165.9 ms	460 MB	7.0	5,283
1 (drift)	0.333	0.82 ms	4.2%	125	22.0 ms	460 MB	12.6	418

Table 6: In-process server telemetry per sweep level. "GC/query" is GC events over server-side query count for the window; "Major GC" is major-collection count per 90 s window; "EL delay" is mean event-loop delay sampled inside the server (the sampler's resolution floor is ≈20 ms, so the 1-user rows read as near-idle); "Vol. ctx/q" is voluntary context switches per query; "Invol. ctx" is involuntary preemptions per window. Sources as in Table 5.

5.1 The Allocation Footprint per Query Is Fixed

GC events per query (0.20–0.23 laptop; 0.29–0.33 VM) and GC pause per query (0.92–1.26 ms laptop; 0.80–1.00 ms VM) are constant across a 50× concurrency range on both machines. Garbage collection tracks *query volume*, not load level: each translated query allocates a fixed amount, and the collector's work scales linearly with queries served. GC pauses account for ≈4–7% of per-query CPU—bounded overhead, not a scaling risk. Consistent with this, peak RSS is stable across the entire sweep on both machines (no growth trend across 8×90 s windows of sustained saturation), evidence against slow leaks or fragmentation under load.

5.2 Emergent Finding: A Persistent Heap-State Transition on the VM

The VM telemetry reveals a discrete runtime state change that a per-request instrument cannot see. Between the 10- and 15-user windows, three things happen simultaneously:

- major-GC activity per window jumps from 34 to 246 (≈7×);
- peak RSS falls from ≈691 MB to ≈456 MB (−34%);
- per-query CPU steps from ≈12.1 ms to ≈17.2 ms (+42%), and process utilization from ≈1.4 to ≈1.9 cores.

Within each state, per-query CPU is extremely flat (1.2% spread across 15–50 users), and GC *pause* per query barely moves (≈0.91 → 0.95–1.00 ms)—the added collection work runs concurrently on background GC threads, which is where the extra ≈0.5 core of utilization goes. Because those threads run on otherwise-idle cores, client-visible throughput changes by only −4% (141.5 → 135.9 req/s). The transition is **persistent**: when the drift check returned load to a single user, the server stayed in the high-GC state (125 major collections in the window, RSS ≈460 MB, 18.4 ms/query versus 13.5 ms in the same window at the start of the run).

The signature—heap shrinkage plus a sustained rise in major-collection frequency at unchanged pause cost—matches the V8 heap-sizing policy trading memory footprint for collection frequency, entering a smaller-heap operating point under sustained allocation pressure and not returning. The laptop server, by contrast, never leaves its low-GC state (8–11 major collections per window, stable RSS) across the identical sweep. Three practical consequences follow: (a) per-query CPU on the VM is *state-dependent*, ≈12 ms or ≈17 ms, and capacity models should budget the high-GC state since sustained production load is what triggers it; (b) the ≈250 MB of RSS returned is real headroom on a memory-constrained host—the trade is not irrational, it is just undocumented tuning surface; (c) heap-size floors (V8 heap flags) are the identified lever if the low-GC operating point is preferred. This finding was only observable because GC is instrumented inside the engine process.

5.3 Latency Under Load Is Event-Loop Queueing

Mean event-loop delay measured inside the server rises monotonically with concurrency—28 → 304 ms on the laptop and 22 → 166 ms on the VM across 1 → 50 users—and moves in near-constant proportion to mean request latency at every level (≈2.9–3.2× on the laptop, ≈2.2–2.4× on the VM once saturated). This locates the latency growth of Section 4 precisely: requests spend their added time *waiting for the single JavaScript thread*, not executing more slowly once scheduled. It is the direct internal counterpart of the external observation that per-query CPU stays flat while p95 grows 11–28×.

5.4 Scheduling and I/O Behaviour

- **Preemption under saturation (VM).** Involuntary context switches rise from 304 to 5,283 per window (17×) as the process climbs toward ≈1.9 cores—the OS scheduler increasingly preempts the busy process—while voluntary switches per query *decline* from 11.2 to 7.0, a mild batching of waits as the event loop stays busier per wakeup.

- **Write I/O is deterministic (VM).** File-system write operations per query are 0.76 at *every* concurrency level—write volume (logging, metrics persistence) is strictly proportional to queries served, with no load-dependent inflation.
- **Write-call coalescing under load (laptop).** On Windows, where the counter tallies write system calls including socket writes, writes per query fall from 2.00 at 1 user to 0.28 at 50 users (7.1×): as the event loop saturates, the runtime serves the same query volume with far fewer, larger write calls. The engine's syscall efficiency *improves* under pressure—an inherited benefit of the runtime's I/O batching.
- **No file reads on the hot path.** File-system read operations by the server process are zero (VM, all levels) or negligible: the query hot path touches the database over sockets and in-memory registries only.

6 Write Operations: CRUD, Transactions, and History

The write path executes every resource mutation through a single versioning gateway that records a cryptographically signed, hash-chained history entry as part of the same transaction. Table 7 reports the measured latency of the write and history surface; every create, update, and delete below produced a signed `ResourceHistory` entry.

Scenario (p95)	Iterations	Laptop SSD	VM NVMe
READ_BY_ID	100	6.22 ms	5.69 ms
CREATE_PATIENT	100	321.77 ms	20.91 ms
CONDITIONAL_CREATE	10	163.47 ms	85.80 ms
CONDITIONAL_UPDATE	10	15.13 ms	75.34 ms
CONDITIONAL_DELETE	5	11.27 ms	65.20 ms
TRANSACTION_1_ENTRY	10	622.03 ms	22.87 ms
TRANSACTION_5_ENTRIES	10	147.54 ms	55.89 ms
TRANSACTION_10_ENTRIES	5	237.02 ms	94.09 ms
BATCH_5_ENTRIES	10	504.92 ms	61.97 ms
BATCH_10_ENTRIES	5	977.39 ms	112.84 ms
INSTANCE_HISTORY	100	13.61 ms	8.63 ms
TYPE_HISTORY	100	17.54 ms	11.72 ms
HISTORY_SINCE	100	20.39 ms	15.67 ms
HISTORY_AT	100	8.56 ms	6.96 ms
HISTORY_SYSTEM	100	12.91 ms	10.69 ms

Table 7: Write, transaction, and history latency. Sources as in Table 3. On the VM the entire write surface sits at 21–113 ms p95 with cost scaling in entry count. Laptop write figures are volatile run-to-run (the same scenario can vary several-fold between same-day runs) because signed writes contend for the 5.94 GB host's memory alongside the co-located database; the low-iteration scenarios (5–10) amplify this. Read and history paths are fast and stable on both tiers.

6.1 Integrity of the Audit Trail

Each mutation records a history entry carrying a full integrity envelope, verified against the live database:

Listing 1: ResourceHistory integrity envelope for a create/update/delete

```
"integrity": {
  "hash":         "60b56d59...248725",   // SHA-256 of the versioned resource
  "previousHash": "52eee5eb...4c29ce",   // chains to the prior entry (tamper-evident)
  "signature":    "3045022057bc...edbc7",// ECDSA P-256 signature
  "keyId":        "<active-signing-key>",
  "algorithm":    "sha256"
}
```

The persist and the signed history entry are committed atomically: a failure to sign or record history rolls the resource mutation back, so a resource never persists without its audit entry. A single create/update/delete lifecycle exercised end-to-end produced exactly one history entry per operation (create → version 1, update → version 2, delete → HTTP 204 followed by HTTP 410 Gone on subsequent read, per FHIR R4 §3.1.0.5).

7 Scalability

The two environments run the identical query workload against corpora differing by 83.6× in clinical-core document count (Table in Section 2). This provides a direct, if hardware-confounded, scaling measurement.

Metric	Laptop SSD	VM	Ratio
Clinical-core documents	77,211	6,458,565	**83.6**×
STRING_SEARCH_EXACT p95	34.13 ms	83.07 ms	2.43×
TOKEN_SEARCH_OBSERVATION p95	44.04 ms	24.45 ms	0.56×
DATE_RANGE_SEARCH p95	53.80 ms	30.37 ms	0.56×

Table 8: Cross-environment scaling. An 83.6× data increase changes indexed-search p95 by at most 2.4× (and *reduces* it on token/date scenarios, where the VM's faster hardware dominates), consistent with $O(\log n)$ index utilization rather than $O(n)$ scanning.

Structural reading. If query cost were dominated by data volume, every scenario would run markedly slower on the 83.6× VM corpus. Instead only the scan-heavier `STRING_SEARCH_EXACT` pays (2.4×), while tightly-indexed token- and date-range scenarios are *faster* on the VM—the corpus scales 83.6× while indexed latency stays within a small constant factor. Taking `STRING_SEARCH_EXACT` at face value, an 83.6× data increase for a 2.4× latency increase is ≈34× better than linear; the token/date scenarios are bounded by hardware, not data, and so cannot isolate the exponent. Hardware differs between the two tiers, so these ratios *bound* rather than isolate the dataset effect—but the direction (indexed latency nearly flat under a two-order-of-magnitude data increase) is unambiguous and is the expected signature of B-tree index utilization.

7.1 Deep-Pagination Invariance

Deep-offset pagination shows no cost penalty on the production-scale VM corpus: `PAGINATED_PAGE_N` (deep offset) p50 equals `PAGINATED_PAGE_1` p50 to within 2% (47.2 vs 48.1 ms). Pagination cost is effectively constant in offset rather than O(offset)—a useful property for dashboard and list workloads.

8 Statistical Reliability

8.1 Coefficient of Variation

Per-scenario coefficient of variation (CV, standard deviation over mean) on the 100-iteration measured windows:

Category	Laptop CV	VM CV
Indexed search (string/token/date)	15–25%	8–14%
Terminology cascades (multi-second)	9–10%	1–3%

Table 9: CV by category. On the small laptop corpus, CV on the 20–70 ms indexed scenarios is dominated by host scheduling noise; the fixed-cost terminology cascades on the VM are highly deterministic (CV 1–3%) because they are recursive value-set traversals of near-constant cost.

8.2 JIT and Cache Warmup Convergence

Coefficient of variation falls sharply from warmup to steady state as the V8 JIT optimizes hot paths and pipeline/value-set caches fill—most dramatically on the VM's cold-cache reverse-chain scenario, whose warmup CV of 404% (driven by the 8.5 s first iteration) collapses to 13% in steady state, an order-of-magnitude-plus tightening.

9 Staged Concurrency Stress

A staged stress test ramps through 15, 25, and 50 concurrent users against a mixed read/write/search scenario pool. Consistent with the saturation profile of Section 4, per-user throughput falls as concurrency rises and latency grows; the error rate is constant across phases and is a property of the scenario mix (a fixed fraction of the pool is negative-test or unsupported input), not of overload.

Env	Phase	Requests	Error rate	p95
VM	1 (15 VU)	69,171	30.0%	163 ms
VM	2 (25 VU)	25,689	30.7%	188 ms
VM	3 (50 VU)	18,711	29.8%	227 ms
VM	overall	113,624	30.1%	193 ms
Laptop	1 (15 VU)	21,935	29.8%	743 ms
Laptop	2 (25 VU)	7,981	29.8%	814 ms
Laptop	3 (50 VU)	5,939	30.4%	913 ms
Laptop	overall	35,908	29.8%	824 ms

Table 10: Staged stress. Sources: `stress-staged-summary.json` (each environment). The ≈30% error rate is flat across a 3.3× concurrency range on both machines, confirming it is scenario-mix-driven. The VM sustains 3.2× the request volume at roughly a quarter of the laptop's p95.

Reading. This is a saturation regime, useful for capacity planning: the knee is visible, per-user throughput collapses as users are added, and latency grows. It should be read as the operating envelope of a single-thread-bound service (Section 4), not as graceful horizontal scaling. The constant ≈30% error rate should be classified (intentional negative tests vs. unsupported queries vs. genuine failures) before it is used to characterize either validation coverage or failure behaviour.

10 Safety-Kernel Absorption

A dedicated benchmark drives an attack corpus (deep chain-depth, oversized value-count, and malformed/operator-injection inputs) alongside valid traffic against a live server on each machine, including a sustained attack-storm phase, and measures rejection correctness and cost directly.

10.1 Correctness

Metric	Laptop	VM
Attack requests issued	23,464	26,838
Absorbed (rejected)	23,464	26,838
Absorption rate	**100%**	**100%**
Hard leaks (attack reached DB)	0	0
Valid queries passed	2,177	2,501
False positives	**0**	**0**

Table 11: Every attack in both corpora was rejected; no valid query was mistakenly blocked. Sources: `safety-kernel/kernel-benchmark-2026-07-18T14-00-48.json` (laptop), `kernel-benchmark-2026-07-18T12-29-22.json` (VM).

Server-side guard counters attribute the rejections to named mechanisms—chain-depth guard (7,707 laptop / 8,955 VM) and value-count guard (1,930 / 2,212)—covering ≈41% of client-observed rejections; the remainder are turned away at the validation layer (malformed input, operator injection, bad tokens), which currently has no per-mechanism counter. Raising counter coverage is a noted instrumentation gap.

10.2 Cost Asymmetry and Storm Behaviour Differ by Tier

The economics of rejection are hardware-dependent and are reported per machine:

Metric (median)	Laptop	VM
Valid query baseline	9.97 ms	18.52 ms
Rejection (chain / value / malformed)	11.4 / 11.8 / 11.4 ms	3.3 / 3.8 / 3.4 ms
Rejection-to-valid cost ratio	0.85–0.88×	**4.9–5.6× cheaper**
Valid traffic under attack storm	37.3 ms (3.74× worse)	13.0 ms (no degradation)
Peak server heap during storm	96.2%	92.3%

Table 12: Cost asymmetry and traffic isolation under storm. On the VM the kernel refuses work before the database is touched and rejections are ≈5× cheaper than valid queries; legitimate traffic sharing the process shows no measurable median degradation under the storm. On the laptop the asymmetry *inverts*—a rejection costs slightly more than this tier's (much cheaper) valid baseline—and valid traffic degrades 3.74× under storm as rejection responses compete for the single thread on a slower core.

Honest limits. Two limits are on the record. First, correct rejection is not denial-of-service immunity: on the laptop tier, a sustained storm degrades legitimate traffic 3.74× at the median, and server heap peaks above 92% on both machines—the single-process deployment remains subject to queueing and memory pressure under volumetric attack (the same single-thread ceiling as Section 4). Second, the gap probe found **seven candidate attack patterns that pass through the guards** and execute as legal-but-expensive queries: 5 on the laptop (median 7.5 ms, max 154 ms) and 4 on the VM including one that ran to the 20.0 s client timeout, plus one server error. What is supported is that no corpus attack reached the database and none crashed the server; the passthrough patterns are enumerated optimization targets for new guards.

11 Optimizer Activity

Server-side optimization counters confirm the query optimizer executes and records work: each full serial benchmark reports `optimizationCounts = { "where->flattenable": 120 }` (the count is reproduced exactly across runs on both machines, consistent with a deterministic scenario mix). The average complexity score is unchanged before and after this rewrite class (`averageComplexity.before == after == 38.25`), indicating the recorded rewrite is score-invariant—either because the score is insensitive to this particular transformation, or because the emitted intermediate representation is already near-canonical. Distinguishing these requires source-level confirmation of what the rewrite does; the measurable fact is that the optimizer runs and logs work on every query.

12 Known SLA Breaches (Optimization Targets)

Reported alongside the passing results, the following scenarios breach their SLA and are optimization targets rather than validated behaviour.

12.1 Stable Breaches (Reproduced on Every Run)

Scenario	SLA	Laptop p95	VM p95	Character
MODIFIER_BELOW	600 ms	858 ms	7,889 ms	terminology cascade (data-scaled)
COMPOSITE_COMBO_CODE_VALUE	600 ms	868 ms	8,059 ms	terminology cascade (data-scaled)
MODIFIER_IN	500 ms	9 ms (pass)	3,473 ms	terminology cascade (data-dep.)
DATE_PREFIX_EQ	200 ms	1,268 ms	377 ms	date-approximate operator
DATE_PREFIX_AP	300 ms	1,222 ms	589 ms	date-approximate operator

Table 13: Stable SLA breaches (laptop SSD run and VM `13-58-39` run; every July 18 serial run reproduces this set). The multi-second terminology cascades are recursive value-set traversals whose cost scales with the corpus, dominating on the VM; the date-prefix operators are slower on the laptop. Low variance on these scenarios (VM CV 1–3%) reflects deterministic cost, not acceptability—an 8-second query is an item to fix.

12.2 Environment- and Client-Dependent Breaches

- **Laptop write-path flicker.** Across three same-day laptop serial runs, each run breaches a *different* subset of write scenarios (`CREATE_PATIENT` 322 ms and `TRANSACTION_1_ENTRY` 622 ms on the SSD run; `BATCH_5/10_ENTRIES` 1.4/3.2 s on the HDD run; `CONDITIONAL_CREATE` 886 ms and `BATCH_10_ENTRIES` 2.4 s on the repeat run), while the VM passes the entire write surface on both runs. The breach set's instability, combined with the 5.94 GB host running the database alongside the server, marks these as host-resource artifacts rather than engine defects—but they are what a deployment on comparable hardware would experience, and are reported as such.
- **Cold-cache reverse chains under the k6 sequential client (VM).** In the serial harness, reverse-chain scenarios pass comfortably on both machines (45–57 ms p95 steady state). Under the k6-driven sequential suite on the VM, the same scenarios breach severely (`REVERSE_CHAIN_SIMPLE` 8.6 s, `_NESTED` 9.4 s, `_ENCOUNTER` 2.6 s, `REVINCLUDE` 1.0 s p95). The serial warmup data (Section 3) shows a one-time 8.5 s cold cost that steady-state traffic amortizes; the k6 sequential pattern evidently re-pays it, indicating the underlying cache does not stay warm across that client's request pattern on the production-scale corpus. Making the reverse-chain cache robust to request pattern is a named optimization target.

13 Appendix A: Depth-Boundary Enforcement

The `HANG_PREVENTION_DEPTH_LIMIT = 11` constraint prevents exponential `$lookup` cascades that could exhaust MongoDB memory. Verified by integration test `tests/FHIR2MongoDB/Integration/complexQueryIntegration-depthBoundary.test.js`:

Depth	Complexity score	Status
1	∼126.5	Allowed
4	∼333.5	Allowed
10	∼750	Allowed
11	<10,000	Boundary pass
12	—	Rejected (<100 ms)
30	—	Rejected (<100 ms)

Table 14: Depth-boundary enforcement: chains at depth 12 and beyond are rejected in under 100 ms with a `FHIRSearchError` (code `too-complex`) before any database work.

14 Appendix B: Circular-Reference Protection

Cycle detection (test `tests/Query/circularReferenceDetection.test.js`) rejects circular reference chains (e.g. `Organization` → `partOf` → `partOf`) in under 100 ms, and terminates a 52-level deep chain in under 500 ms, using an $O(n)$ visited-reference set. No circular query hangs or exhausts memory.

15 Appendix C: Registry Micro-Latency

Schema and terminology resolution is served from in-memory registries (test `tests/Refactoring/registry-stress.test.js`):

Operation	Iterations	Avg latency
SearchRegistry lookup	1,000	0.084 ms
TerminologyRegistry lookup	1,000	0.092 ms
Mixed registry operations	10,000	0.0 MB heap delta

Table 15: Sub-100-microsecond schema resolution with zero heap growth over 10,000 operations.

16 Appendix D: Transaction Rollback (ACID)

Transaction rollback is verified across 12 scenarios (test `tests/Query/transactionRollback.test.js`), all passing: PUT/DELETE rollback restores the exact prior state and version; multi-entry bundles roll back atomically on any entry failure; independent operations execute in parallel; and circular and self-referencing dependencies are detected. A PUT that fails mid-transaction restores the original resource including its `meta.versionId`, confirming MongoDB `abortTransaction()` semantics are correctly wired through the executor.

17 Appendix E: Query Cost Model

The k6 request-timing breakdown confirms that server processing dominates request latency (VM raw stream, 9,531 requests):

Phase	Mean time	Share
Server processing (`http_req_waiting`)	277.17 ms	99.9%
Network TX (`http_req_sending`)	0.04 ms	0.01%
Network RX (`http_req_receiving`)	0.26 ms	0.09%

Table 16: Request-timing decomposition (VM, `k6-raw-nvme.json`). Over 99% of latency is server-side; network overhead is negligible on the loopback path, so the latency figures throughout this report are engine cost, not transport cost.

18 Raw Data References

All figures are reproducible from the files below. Paths are relative to the repository `benchmark-results/` tree, `[VM]` denotes the virtual-machine results tree, and each serial-run file exists in a full `confidential/` and a sanitized `public/` variant:

File	Content
concurrency-sweep/sweep-2026-07-18T10-19-48.json	Laptop sweep + telemetry (Secs. 4, 5)
[VM]/concurrency-sweep/sweep-2026-07-18T10-37-57.json	VM sweep + telemetry (Secs. 4, 5)
benchmark-2026-07-18T10-20-50-{ssd,hdd}.json	Laptop latency / SLA / CRUD
benchmark-2026-07-18T12-03-30.json	Laptop serial repeat run
[VM]/benchmark-2026-07-18T13-58-39.json	VM latency / SLA / CRUD
[VM]/benchmark-2026-07-18T11-18-10.json	VM serial repeat run
benchmark-2026-07-18T11-37-39.json, [VM]/benchmark-2026-07-18T12-33-25.json	k6-driven sequential suites
stress-staged-summary.json, [VM]/stress-staged-summary.json	Staged stress (Section 9)
safety-kernel/kernel-benchmark-2026-07-18T14-00-48.json	Laptop Safety-Kernel run (Section 10)
[VM]/safety-kernel/kernel-benchmark-2026-07-18T12-29-22.json	VM Safety-Kernel run (Section 10)
k6-raw-{ssd,hdd}.json, [VM]/k6-raw-nvme.json	Raw k6 NDJSON timing

Table 17: Benchmark source files (July 18, 2026, build `21e4d9f`).

19 Conclusions

1. **Low, predictable indexed latency:** p95 of 24–83 ms on core search scenarios across both tiers, with tail ratios of 1.1–1.3×; over 99% of request latency is server-side.

2. **Per-query CPU is flat in concurrency:** a controlled 1–50-user sweep measured against the server process shows no amortization and no super-linear growth on either machine. The engine is single-thread-bound at ≈25 ms/query (laptop) and ≈12–17 ms/query (VM, heap-state-dependent); concurrency buys queueing latency, not throughput, past a per-core ceiling (≈57–63 and ≈135–147 req/s). Higher throughput requires horizontal scaling.

3. **The runtime's internals are now observable and behave lawfully:** in-process telemetry shows a fixed allocation footprint per query (≈1 ms GC pause per query at every concurrency level on both tiers, no RSS growth under sustained saturation), latency under load located at the event loop, deterministic write I/O per query on Linux, and improving syscall efficiency under pressure on Windows.

4. **One emergent runtime phenomenon is on the record:** the VM server exhibits a discrete, persistent heap-management state transition under sustained load—major-GC activity ×7, RSS −34%, per-query CPU +42%, client throughput −4%—with hysteresis (it does not revert when load drops). Capacity planning should budget the high-GC state; heap-size flags are the identified lever.

5. **Correct, signed write path:** create, conditional update/delete, transactions, batches, and history all execute, each producing an atomically-committed, ECDSA-signed, hash-chained audit entry; transaction rollback restores exact prior state across 12 verified scenarios. The VM holds the entire write surface at 21–113 ms p95; laptop write latency is volatile under that host's memory pressure and is reported as such.

6. **Logarithmic data scaling:** an 83.6× resident-dataset increase changes indexed p95 by at most 2.4× (and reduces it where hardware dominates), with constant-cost deep pagination—consistent with $O(\log n)$ index utilization.

7. **Bounded, correct rejection with tier-dependent economics:** 100% absorption of 50,302 attack requests across both machines with zero false positives. Rejections are ≈5× cheaper than valid queries on the VM, where storm traffic shows no measurable degradation of valid queries; on the laptop the asymmetry inverts and storms degrade valid traffic 3.7×. Seven guard-passthrough patterns (one reaching 20 s) are enumerated for new guards.

8. **Named optimization targets:** terminology-expansion cascades (up to ≈8 s on the production corpus), two date-prefix operators, laptop write-path volatility, guard passthroughs, and the request-pattern-sensitive reverse-chain cache are the open items, reported explicitly for remediation.

Benchmark date: July 18, 2026. Build: `21e4d9f`*. Resident datasets counted from the live databases (laptop 77,211 clinical-core / 743 patients; VM 6,458,565 clinical-core / 10,322 patients). All numerical claims reproducible via* `node -e` *against the cited JSON paths; CPU figures against the server-process sampling and GC/event-loop/I/O figures against the in-process* `serverGc/serverIo/serverEventLoopMeanMs` *records in the sweep files.*

References

1. Möbius, C., Dargie, W., & Schill, A. (2013). "Power Consumption Estimation Models for Processors, Virtual Machines, and Servers." *IEEE Transactions on Parallel and Distributed Systems*, 25(6), 1600–1614.

2. Khan, K.N., et al. (2018). "RAPL in Action: Experiences in Using RAPL for Power Measurements." *ACM TACO*, 3(2), 1–26.

3. Hackenberg, D., et al. (2013). "Power Measurement Techniques on Standard Compute Nodes." *IEEE ISPASS*, 194–204.