CASE STUDY

The Hidden Cost of Standing Still — A Genetic Testing Lab's Platform Overhaul

How NonStop Modernized a Genetic Testing Lab's Core Platform

Clinical Genomics · Platform Modernization · Architecture Engineering

Executive Summary

What was the business suffering?

- Every new feature took weeks longer than it should — the old system made every change risky
- Nothing could be updated independently — fixing one thing meant retesting everything
- Database changes were done manually in live production — high risk, frequently avoided
- When something broke, no one could see it until it was already a problem
- Lab staff re-keyed the same information on every single order, every day

What did it cost the Business?

- Slower product improvements = slower ability to win and onboard new hospital clients
- Releases were infrequent and stressful — the team shipped less, clients waited longer
- Features that required any data structure change simply didn't get built
- Downtime was diagnosed late — SLA risk, clinical delays, staff productivity loss
- Order errors, correction queues, amended reports, and delayed billing — all traced back to manual re-entry

What NonStop Built?

- Modernized the platform piece by piece — the lab never stopped operating during the transition
- Each part of the system (orders, billing, reporting) now works and ships independently
- Database changes are now safe, tracked, and reversible — like any other code update
- Live dashboards show exactly what the system is doing at all times
- Order search across 70,000 records now returns in under a second

What the Business Gained

- New hospital onboarding dropped from weeks to days
- Feature delivery speed roughly doubled
- Order defects dropped to near zero — fewer corrections, fewer resubmissions, cleaner billing
- Reports reached clinicians faster — directly protecting revenue cycle TAT
- Deployments went from high-stakes events to routine, low-risk operations
- Zero disruption during the full platform migration

The Real Cost of a Platform You Can't Move

For a genetic testing lab, the software platform isn't peripheral infrastructure. It's the operational backbone, order intake, accessioning, reporting, and billing; every clinical and commercial workflow runs through it. When that backbone is built on a decade-old monolith that can't evolve without risking what already works, the cost isn't visible in a single outage. It's visible in everything that moves slower than it should.

Our client was operating on a .NET v2 monolith. The system processed orders, generated reports, and handled billing; it worked. But it was architecturally frozen. Any new feature had to be written in .NET v9 while maintaining backward compatibility with the legacy codebase. The entire application shipped as a single deployment unit. Schema changes were manual database entries in production. Debugging meant SSH access to read raw server logs. There was no safe way to deploy incrementally, no observability layer, no versioned data evolution.

Every product decision, every operational improvement, every new hospital onboarding was filtered through the same question: Can the legacy architecture support this without breaking what's already running? The answer was increasingly no, or at least, not without disproportionate engineering effort.

> **"The platform wasn't broken. It was frozen. Every change carried the risk of breaking backward compatibility with a decade of accumulated logic. The architecture had become the constraint on the business."**

What the Engineering Team Was Actually Dealing With

Backward Compatibility as a Development Tax

The .NET v2 codebase meant any new capability had to be written in .NET v9 while preserving full backward compatibility with legacy modules. This wasn't a team velocity problem; it was a compounding architectural debt. Each new feature carried the overhead of ensuring it didn't break a decade of accumulated logic. Development cycles stretched not because the engineers were slow, but because the architecture demanded caution at every integration point.

Monolith Coupling Every Release

A single codebase meant a single deployment artifact. A bug fix in reporting couldn't ship without regression-testing order entry. Feature work in one domain was blocked by release readiness in another. The team couldn't deploy independently, couldn't isolate failures, and couldn't scale components that needed scaling without scaling everything. Every release was an all-or-nothing event.

Database Changes as Manual Production Interventions

Schema changes were manual entries executed directly in production. No migration scripts, no version control on the data layer, no rollback path. Every structural change to the database was a one-way bet. The team avoided necessary schema evolution not because it wasn't needed, but because the risk of executing it was disproportionate to the change itself. The data layer was stuck alongside the application layer.

Zero Observability

No monitoring layer existed. Diagnosing an API failure, a performance degradation, or an order processing error required direct server access to read raw log files. Every debugging session pulled an engineer off feature work and into infrastructure forensics. The team couldn't see what the system was doing; they could only investigate after something had already gone wrong, manually, one log file at a time.

No Safe Deployment Path

Without an incremental deployment capability, every release carried full production risk. There was no canary path, no percentage-based rollout, no automated rollback. A regression in any module meant rolling back the entire application. The deployment model discouraged frequent releases, which in turn slowed the feedback loop between development and production.

No Data Persistence Layer for Reusable Context

The application had no mechanism for persisting and reusing frequently entered data, physician profiles, provider credentials, or common order templates. Every workflow starts from a blank state. This wasn't a UI shortcoming; it was a data architecture gap. Without a persistence and retrieval layer for reusable entities, the application couldn't learn from its own usage patterns, pushing the redundancy cost onto users for every order, every day.

What We Built and Why

The diagnosis was clear: the bottleneck wasn't engineering talent or team size. It was that a monolithic, decade-old architecture was forcing modern requirements through legacy constraints at every layer, application, data, deployment, and observability. The fix wasn't patching the existing system. It was rebuilding the platform on an architecture where each layer could evolve independently, without disrupting the clinical operations running on the production system.

We structured the modernization across six architectural layers, each addressing a specific engineering failure point.

Micro-Frontend Architecture for Zero-Disruption Migration

The single biggest architectural risk in any platform modernization is the cutover. A big-bang migration, shutting down the old system and switching to the new, is operationally unacceptable for a lab processing clinical orders daily.

We used micro-frontend architecture to decompose the monolithic UI into independently deployable frontend modules. Each workflow surface, order entry, search, and reporting was rebuilt and swapped in individually, behind the same application shell. The lab team experienced gradual, incremental improvement. At no point did operations stop. At no point did users face a full platform switch. The old and new coexisted in production until the migration was complete.

> **"The migration principle: never ask a clinical lab to stop operating so you can modernize their platform. The old and new systems coexist in production. You swap surfaces one at a time, and the user barely notices."**

Microservices Decomposition

The monolith was decomposed into domain-specific microservices: order management, reporting, billing, search, and user management, each independently deployable, testable, and scalable. A bug fix in one service ships without regression-testing the others. Feature teams own their domain end-to-end. Failure in one service is isolated; it doesn't cascade across the platform.

This decomposition is what unlocked the downstream improvements: independent deployment, targeted scaling, and faster feature cycles. Without it, every other modernization would have been bolted onto the same monolithic constraint.

Versioned Migration Scripts for Data Layer Evolution

Manual database entries were replaced with versioned, code-reviewed migration scripts. Every schema change is now tracked in version control, executed through an automated pipeline, and reversible. Database evolution became a pull request, reviewed, tested, and deployed like any other code change.

Feature development that previously stalled because it required a schema change, a new table, a column addition, or an index, now moves at the same pace as application code. The data layer is no longer the bottleneck.

Grafana-Based Observability Stack

Server-access debugging was replaced with a full observability layer. Grafana dashboards surface API health, response times, error rates, throughput, and service-level performance in real time. Engineers diagnose from dashboards, not SSH sessions. Degradations are visible before users report them. Alert thresholds trigger notifications, not manual log inspections.

For a platform running clinical order workflows, this isn't a developer convenience; it's operational safety. The team can see what the system is doing at all times, without touching production infrastructure.

Canary Deployment Pipeline

All-or-nothing releases were replaced with a canary deployment pipeline. New versions roll out to a small percentage of production traffic first. Automated health checks validate the release against real-world load. If metrics degrade, rollback is instant and automatic. If they hold, the release expands incrementally to full traffic.

This changed the team's relationship with deployment. Releasing stopped being a risk event and became a routine operation. Feature delivery accelerated because the cost of shipping and the risk window shrank to near zero.

Elasticsearch for High-Volume Data Retrieval

Order search was offloaded from the primary database to Elasticsearch. Seventy thousand order records now return within twenty milliseconds. The previous approach, direct relational queries against the production database, was slow under load and competed with transactional operations for database resources.

With Elasticsearch handling read-heavy search workloads, the primary database is reserved for writes and transactional integrity. Search performance became independent of database load, a separation of concerns at the data layer that directly improved both operational speed and system stability.

By the Numbers

Metric	Before	After
Architecture	Monolith (.NET v2)	**Microservices + micro-frontends**
Feature development cycle	Weeks (backward-compat overhead)	**~50% reduction**
Deployment model	All-or-nothing releases	**Canary pipeline with auto-rollback**
Database schema changes	Manual production entry	**Versioned migration scripts**
System observability	SSH + raw server logs	**Grafana real-time dashboards**
Order search (70K records)	Seconds (direct DB query)	**~20 milliseconds (Elasticsearch)**
Migration disruption	N/A (never attempted)	**Zero downtime (micro-frontend swap)**
Service isolation	None (single deployment unit)	**Independent per domain**

The Impact

- **Turnaround time dropped, directly protecting revenue.** Cleaner data intake, driven by auto-populated fields, schema validation, and the elimination of manual re-keying errors, meant fewer orders stalled in correction queues. Reports reached clinicians faster.
- **Report accuracy improved at the source.** Downstream reports carried cleaner data from the first step. Fewer corrections, fewer amended reports, fewer billing resubmissions, each of which had been a direct drag on revenue cycle efficiency.
- **New hospital onboarding accelerated from weeks to days.** The microservices architecture and configurable integration layer turned new client onboarding from a multi-week engineering project into a configuration task.
- **Ordering defects dropped to near zero.** Orders that entered the LIS were structurally complete and schema-valid. The reduction in order-level defects reduced rework, support tickets, and the hidden cost of clinical staff time spent chasing data corrections.
- **The feature velocity roughly doubled.** Microservices removed cross-domain coupling. Migration scripts eliminated manual database intervention. Development cycles dropped by approximately fifty percent.
- **Deployment became routine, not a risk event.** Canary releases replaced all-or-nothing deployments. The team ships continuously with automated rollback.

- **The data layer stopped being a bottleneck.** Versioned migration scripts made schema evolution a normal part of the development workflow. Features that required database changes no longer stalled in a manual intervention queue.
- **Debugging moved from hours to minutes.** Grafana-based observability gave the team real-time visibility into every service, every API, every performance metric, without touching production servers.
- **Migration happened with zero operational disruption.** Micro-frontend architecture allowed surface-by-surface modernization while the lab continued processing orders on the live system.
- **Search performance became an operational asset.** Elasticsearch delivering seventy thousand records in twenty milliseconds transformed order lookup from a slow database query into an instant retrieval tool, directly improving intake, QA, and support workflows.

> **"The engineering team went from managing a frozen system, where every change was a backward-compatibility risk calculation, to operating a platform where services evolve independently, deployments are routine, and the architecture compounds capability instead of constraining it."**

A lab's engineering leadership shouldn't spend its time navigating backward-compatibility constraints from a decade-old codebase, manually entering schema changes in production, or reading server logs through SSH to diagnose a failed API call. That's not engineering, it's maintenance disguised as progress.

NonStop rebuilt the platform so the architecture could finally compound instead of constrain. The result wasn't just a modern stack. It was an engineering team that could ship features, evolve the data layer, deploy safely, and scale the platform, all without asking permission from the legacy system they left behind.

Interested in building the same capabilities for your lab?
nonstopio.com | Clinical Genomics & Life Sciences Engineering