# Their Core Banking Ran on COBOL. Nobody Could Prove What It Did.

A Tier-1 Thai commercial bank approached us with the problem that every institution running a mainframe eventually confronts. The system worked. It had worked for decades. But the engineers who wrote it had retired, the documentation described an architecture that no longer existed, and **every proposed change carried a risk that nobody in the building could quantify.** The bank was not paralysed by technology. It was paralysed by the absence of evidence.

We migrated the application estate from COBOL to Java and Spring, moved batch processing off the mainframe onto distributed infrastructure, retired Oracle and DB2 in favour of PostgreSQL, decomposed the monolith into domain services, and reconstructed the documentation from the source code itself. Five workstreams, one estate, one programme — governed throughout by a single discipline: **prove that the new system does exactly what the old one did, and produce the evidence an auditor can inspect without taking our word for anything.**

This document describes how that was done. The client is not named and the metrics are not published, because the engagement is bound by non-disclosure. What transfers is the method, and the method is what follows.

## THE FIVE WORKSTREAMS

01

### COBOL → Java / Spring

Deterministic translation of the application estate. The same source yields the same output on every run — not a probabilistic rewrite that an auditor is asked to accept on faith.

02

### Mainframe offload to distributed

Batch workloads moved onto commodity infrastructure while the system of record remained intact and authoritative throughout the entire transition.

03

### Oracle / DB2 → PostgreSQL

Full data-tier migration. Stored procedures, triggers, collations and access paths rebuilt and reconciled row by row rather than lifted blind.

04

### Monolith → microservices

Decomposed along domain boundaries recovered from the code itself, not from the organisation chart that the last architecture diagram happened to be drawn against.

05

### Documentation reconstruction

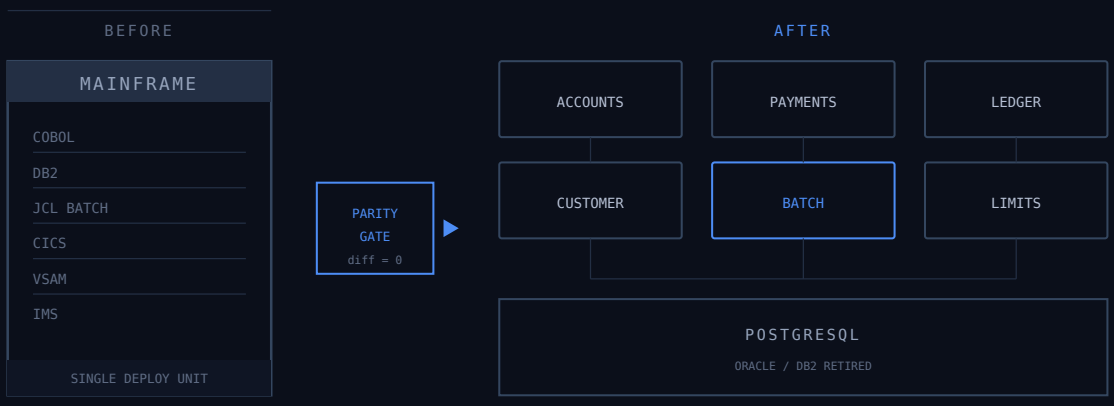
Specifications rebuilt from the source. What the code actually does, written down — for the first time in the estate's operational history.

+

### Behavioral equivalence testing

The discipline beneath all five. Every migrated path replayed against the original and compared. The evidence, not the assurance.

BEFORE AND AFTER




FOUR MIGRATIONS ON ONE ESTATE · EXECUTED AS A SINGLE PROGRAMME

## WHY BANKS DO NOT MIGRATE

The barrier to mainframe modernisation is almost never technical. It is evidentiary. A bank's model risk function, its internal audit team, and eventually its examiner will all ask the same question, and **"the vendor says it is equivalent" is not an answer that any of them accept.** The migration is the easier half of the problem. Proving it is the half that kills programmes.

This matters more than it might appear. A modernisation programme that cannot produce evidence does not merely carry technical risk — it carries the risk that the institution will be unable to defend the change to its own governance functions after the fact. Programmes are cancelled at month eighteen not because the code does not work, but because nobody can demonstrate that it works in the way the previous system worked. Every subsequent conversation becomes a negotiation over confidence rather than a review of facts.

### THE FEAR

Silent behavioural drift. A rounding rule, an edge case, an undocumented branch — wrong in production, discovered by a customer rather than by a test.

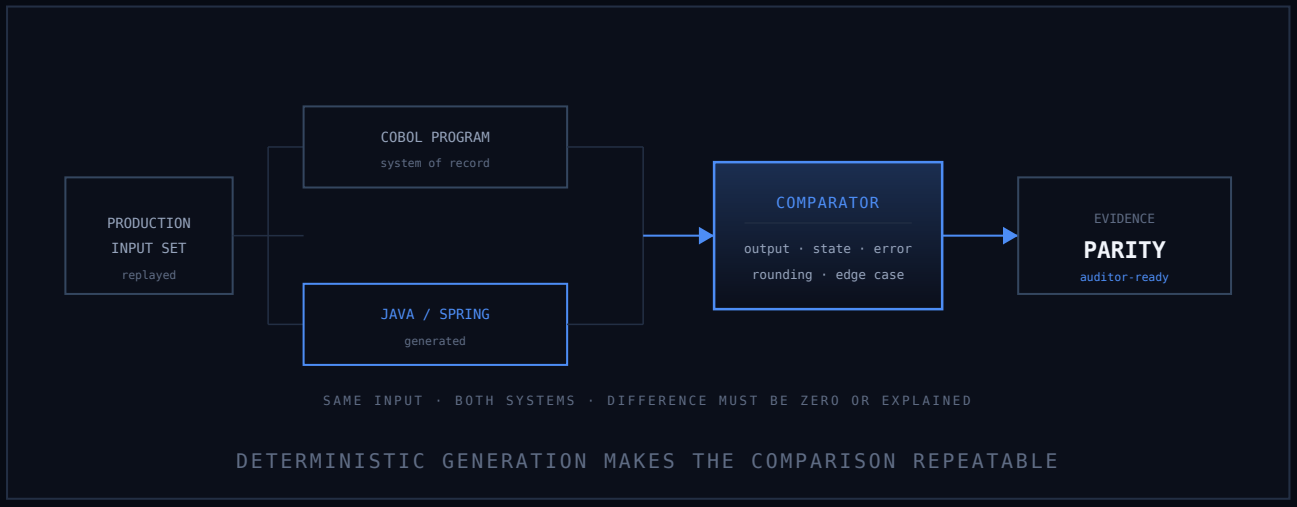
### THE ANSWER

Replay both systems against identical production inputs. Compare outputs, state transitions and error behaviour. The difference is zero, or it is explained in writing.

### THE ARTIFACT

A parity record for every migrated path. Not a status report — evidence an auditor can inspect independently of the delivery team.

## THE EQUIVALENCE DISCIPLINE




DETERMINISTIC GENERATION MAKES THE COMPARISON REPEATABLE

The comparator is deliberately unforgiving. It does not compare the final output alone; it compares intermediate state, the sequence of database writes, the precise text and code of any error raised, and the behaviour of the path when given deliberately malformed input. A migration that produces the right answer by a different route is not equivalent, because the route is what the next change will be built upon.

## WHY DETERMINISM IS THE WHOLE ARGUMENT

> Most AI-assisted migration produces different code on different runs. For a marketing team that is a feature. For a bank's model risk function it is disqualifying — you cannot validate what you cannot reproduce.
>
> Our generation is deterministic. The same COBOL yields the same Java, every time. That single property is what converts a migration from a leap of faith into an auditable engineering process, and it is the reason the equivalence evidence carries any weight at all.

The practical consequence is worth stating plainly. When a reviewer challenges a specific transformation eleven months after it was made, the team can regenerate that artefact from the same source and demonstrate that the output is byte-identical to what was deployed. Under a probabilistic approach, that question has no answer at all — the code that would be produced today is simply not the code that was produced then, and the difference between the two cannot be attributed to anything in particular.

WORKSTREAM 01

# COBOL to Java and Spring

The application estate was the accumulated record of decades of decisions, most of them undocumented and many of them deliberate. The task was not to write better code. It was to carry every one of those decisions forward without knowing in advance which of them mattered.

### WHAT WE ACTUALLY DID

Translation proceeded program by program rather than by subsystem, because programs are the unit at which behaviour can be independently verified. Each program was parsed into an intermediate representation that captured control flow, data division structure and arithmetic intent, and Java was generated from that representation rather than from the COBOL text directly. Spring provided the service scaffolding, transaction boundaries and dependency management around the generated logic.

Critically, the generated code was not treated as a draft for engineers to improve. Improving it by hand would have broken the property that made the whole approach defensible — that the output is a reproducible function of the input. Where the generated code was genuinely inadequate, the generator was corrected and the artefact regenerated, so that the corrected behaviour applied uniformly across every program exhibiting the same pattern rather than only where an engineer happened to notice it.

### THE CHALLENGES WE ENCOUNTERED

- **Packed decimal arithmetic.** COBOL's COMP-3 representation and its rounding conventions do not correspond to any Java primitive. Naive translation to `double` introduces errors of a fraction of a cent per operation — invisible in testing, material across a settlement run of millions of transactions, and precisely the class of defect a regulator asks about.
- **GO TO and fall-through control flow.** Decades-old programs contained control structures with no clean structured equivalent. Restructuring them for readability would have changed behaviour in edge cases that nobody had documented and that only production traffic would ever reveal.
- **Implicit data-division dependencies.** Working-storage was reused across paragraphs in ways that created dependencies invisible from the code's apparent structure. A variable set in one branch was read in another that appeared entirely unrelated.
- **Dead code that was not dead.** Paragraphs that appeared unreachable by static analysis were reached in practice through dynamic invocation, and removing them would have failed silently and only under specific conditions.
- **Error handling by convention rather than declaration.** Failure was signalled through status flags and shared state rather than exceptions, and the calling code's interpretation of those flags was itself undocumented behaviour that downstream systems had come to depend upon.

### HOW WE RESOLVED THEM

- All fixed-point arithmetic was mapped to `BigDecimal` with rounding mode and scale set explicitly per operation, derived from the COBOL PICTURE clause rather than inferred. No arithmetic operation was permitted to rely on a language default.
- Irregular control flow was preserved structurally rather than rewritten. The generated Java is less elegant than a human would write, and that is the correct trade: elegance was not the requirement, and the readable version is the one whose behaviour nobody can vouch for.
- Data-division dependencies were recovered through combined static and dynamic analysis, then encoded explicitly in the generated code so that the dependency became visible rather than emergent.
- No code was removed on the basis of static analysis alone. Reachability was confirmed against production execution traces, and anything not positively demonstrated to be dead was carried forward.
- Status-flag conventions were preserved exactly at service boundaries, with the exception-based idiom introduced only inside a service where the convention could not leak to a caller.

### HOW WE PROVED IT WORKED

Every translated program was executed against the same production input set as its COBOL predecessor, and the comparison covered final output, intermediate state, database write sequence and error behaviour. Arithmetic paths received additional scrutiny: inputs were generated specifically to exercise rounding boundaries, and any divergence at the smallest representable unit was treated as a failure requiring explanation, not as noise within tolerance. **No program was accepted on the basis of sampling.**

RISK AVOIDED

A cent-level rounding divergence entering a settlement path and being discovered downstream, weeks later, by a customer or a reconciliation break rather than by a test — with no way to determine how many transactions had already been affected.

WHAT A US BANK SHOULD TAKE FROM THIS

Ask any migration vendor what their generated code does with COMP-3 rounding, and ask them to show you the evidence rather than describe the approach. The answer, and whether they can produce an artefact rather than an assurance, tells you what kind of programme you are buying.

WORKSTREAM 02

# Mainframe Offload to Distributed

The batch estate carried the bank's overnight processing: settlement, interest accrual, statement generation, regulatory extracts. It ran within a window that had been tuned over years and that every downstream system had quietly come to depend upon.

## WHAT WE ACTUALLY DID

Jobs were moved off the mainframe in dependency order, beginning with those at the leaves of the job graph and working inward. The system of record remained on the mainframe throughout, which meant that at every point in the transition there was exactly one authoritative source of truth. Offloaded jobs read from and wrote to that source through a controlled interface, so no job ever operated against a copy whose freshness could be questioned.

## THE CHALLENGES WE ENCOUNTERED

- **The job graph was not what the scheduler said it was.** Declared dependencies were incomplete. Real dependencies ran through shared datasets, through timing assumptions, and in several cases through the fact that one job had simply always finished before another started.
- **The batch window was a hard external constraint.** Downstream systems, some outside the bank's control, expected outputs at particular times. A distributed job that completed correctly but late was, operationally, a failed job.
- **Restart semantics differed fundamentally.** Mainframe checkpoint and restart behaviour has no direct equivalent in a distributed scheduler. A job that failed halfway and restarted cleanly on the mainframe could, naively reimplemented, reprocess records it had already committed.
- **EBCDIC to ASCII conversion affected sort order.** Collation sequence differs between the two encodings, so any job whose output depended on sort order produced correctly-converted records in a different sequence — and downstream consumers depended on that sequence.

## HOW WE RESOLVED THEM

- The true job graph was reconstructed from execution logs rather than from scheduler configuration, exposing implicit dependencies before they could be discovered by breaking them.
- Every offloaded job was required to demonstrate its completion time against the existing window under production-scale data before it was permitted to carry traffic. Timing was treated as a correctness property, not a performance target.
- Restart semantics were rebuilt explicitly through idempotent processing and durable checkpoints, so that reprocessing a record was inert rather than duplicative — and this was verified by deliberately killing jobs mid-run.
- Collation was pinned explicitly wherever sort order was observable downstream, and the resulting sequence was compared against mainframe output record by record.

### HOW WE PROVED IT WORKED

Each offloaded job ran in parallel with the mainframe original against the same input, on production-scale data, for a sustained period before any traffic moved. Outputs were compared record by record, including sequence. Jobs were killed at randomly selected points during these runs and restarted, to confirm that the restart produced identical output to an uninterrupted execution. **Parallel running continued until the evidence was boring** — that is, until repeated runs produced no divergence of any kind.

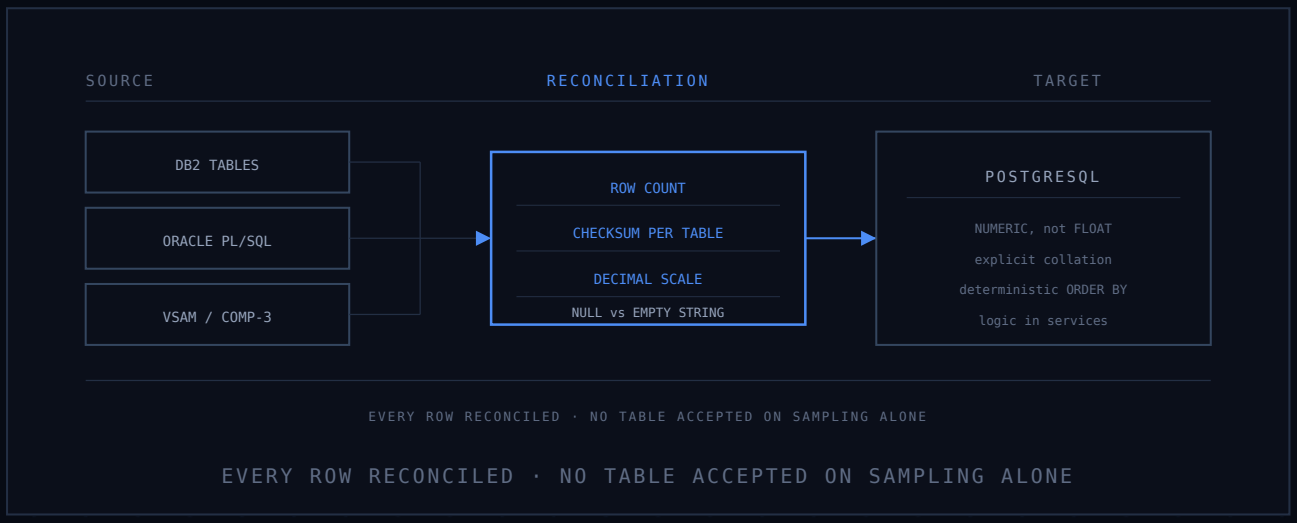
> **RISK AVOIDED**
> A batch window overrun cascading into downstream systems and regulatory extracts — the failure mode that turns a technical incident into a reportable one, and that is discovered by counterparties rather than by the bank.

> **WHAT A US BANK SHOULD TAKE FROM THIS**
> Your batch window is a contract with systems you may not control. Any offload plan that treats completion time as a performance concern rather than a correctness requirement has not understood what it is migrating.

WORKSTREAM 03

# Oracle and DB2 to PostgreSQL

The data tier held the bank's record of truth, along with a substantial quantity of business logic that had accumulated in stored procedures and triggers over many years — logic that no application-level inventory acknowledged.




EVERY ROW RECONCILED · NO TABLE ACCEPTED ON SAMPLING ALONE

THE CHALLENGES WE ENCOUNTERED

- **Empty string is not NULL, except in Oracle.** Oracle treats the empty string as NULL; PostgreSQL does not. Every predicate touching a nullable text column was therefore a potential behavioural change, and the affected columns numbered in the thousands.
- **Numeric type semantics.** Oracle's NUMBER and DB2's DECIMAL do not map cleanly onto a single PostgreSQL type. The wrong choice reintroduces exactly the floating-point rounding problem that the application migration had been careful to eliminate.
- **Business logic embedded in the data tier.** Stored procedures and triggers encoded rules that existed nowhere else. Some were actively maintained; some had not been touched in fifteen years and were still firing on every write.
- **Non-deterministic ordering.** Queries with `ORDER BY` on non-unique columns returned rows in an order that was stable in practice on the source engine and different on PostgreSQL. Application code, and in several cases downstream reports, depended on that incidental stability.
- **Collation and locale.** Sort order for text differed between engines, affecting comparison results, index behaviour and any output whose sequence was observable.
- **Isolation-level behaviour.** The engines differ in their default handling of concurrent reads and writes. Code written against one engine's behaviour can exhibit races under another's without any code change at all.

## HOW WE RESOLVED THEM

- Every nullable text column was audited for empty-string semantics, and predicates were made explicit rather than left to engine behaviour. This was tedious, unglamorous and non-negotiable.
- All monetary and quantity columns were mapped to `NUMERIC` with explicit precision and scale. No monetary value was permitted to touch a floating-point type at any point in the stack.
- Data-tier logic was inventoried in full and then triaged: rules with genuine business meaning moved into services where they became testable and visible; rules that were vestigial were removed only after their non-execution had been demonstrated against production traffic.
- Every `ORDER BY` was made deterministic by appending a unique tiebreaker. Where downstream consumers depended on incidental order, that order was reproduced explicitly and documented as a requirement rather than left as an accident.
- Collation was pinned explicitly per column rather than inherited from the database default.
- Isolation levels were set explicitly per transaction, and concurrent access paths were tested under deliberately induced contention rather than under the benign conditions of a test environment.

## HOW WE PROVED IT WORKED

Migration was verified by full reconciliation rather than by sampling: row counts, per-table checksums, and column-level comparison of decimal scale and precision. Query results were compared between engines for every access path the application exercised, including ordering. Concurrency behaviour was tested under induced contention. **No table was accepted on the basis of a sample, and no access path was accepted on the basis of a single-threaded test.**

RISK AVOIDED

A silent NULL-semantics change altering the result of a balance or eligibility predicate — producing an answer that is wrong, plausible, and therefore not noticed until a reconciliation break or a customer complaint forces someone to look.

WHAT A US BANK SHOULD TAKE FROM THIS

The licence saving is what gets the migration funded. The empty-string semantics are what determine whether it succeeds. If a vendor's proposal discusses the first at length and the second not at all, the proposal is a commercial document rather than an engineering one.

WORKSTREAM 04

# Monolith to Microservices

Decomposition is where modernisation programmes most often inflict damage. Boundaries drawn incorrectly produce a distributed system with all of the monolith's coupling and none of its transactional guarantees — a strictly worse outcome than the estate it replaced.

## WHAT WE ACTUALLY DID

Boundaries were derived from the code and from production behaviour, not from the organisation chart. We analysed which data was written together within a single transaction, which paths shared state, and which entities changed together over time. Where the data said the boundary lay in one place and the org chart said another, the data won.




## THE CHALLENGES WE ENCOUNTERED

- **Transactional integrity across new boundaries.** Operations that had been a single ACID transaction inside the monolith now spanned services. Money movement cannot be eventually consistent.
- **Shared mutable state.** The monolith passed state through shared structures in ways that crossed every proposed boundary, and each crossing was a decision about ownership that had never previously needed to be made.
- **Chatty decomposition.** An early boundary proposal produced a call pattern that would have turned one in-process operation into a large number of network calls, converting a latency non-issue into a latency problem and a reliability problem simultaneously.
- **Failure modes that did not previously exist.** In a monolith, a function call does not time out. Across a service boundary it does, and every such call introduced a partial-failure state that the original code had no concept of and therefore no handling for.

### HOW WE RESOLVED THEM

- Boundaries were drawn so that operations requiring atomicity remained within a single service. Where a genuine cross-service transaction was unavoidable, it was implemented as an explicit saga with compensating actions and a reconciliation path — never as an implicit assumption of eventual agreement.
- Ownership of every piece of shared state was assigned explicitly to exactly one service, with all other access mediated through that owner's interface.
- The chatty proposal was rejected and the boundary redrawn. Correct decomposition is measured by call patterns under production load, not by the tidiness of the resulting diagram.
- Every cross-service call was given explicit timeout, retry and idempotency semantics, and the partial-failure paths were tested by deliberately injecting failures rather than by assuming the network would behave.

### HOW WE PROVED IT WORKED

Decomposition proceeded under a strangler pattern with rollback available at every phase. New services ran in shadow mode — executing against live traffic and producing results that were compared but not served — until parity was demonstrated over a sustained period. Only then was traffic shifted, incrementally, with the legacy path retained and armed. **At no point was there a single irreversible cutover event.** Failure injection confirmed that partial failures degraded predictably rather than silently.

RISK AVOIDED

A distributed monolith: the coupling of the original system combined with the failure modes of a network, and the loss of the transactional guarantees that made the original safe. This is the most common outcome of bank decomposition programmes and it is effectively irreversible once shipped.

WHAT A US BANK SHOULD TAKE FROM THIS

If a proposed service boundary map matches your organisation chart, it was drawn from the org chart. Boundaries recovered from transactional behaviour rarely resemble reporting lines, and the mismatch is a signal worth acting on.

WORKSTREAM 05

# Documentation Reconstruction

The bank had documentation. It described a system that had not existed for years. This is the ordinary condition of a long-lived estate, and it is more dangerous than having no documentation at all, because confidently wrong documentation is acted upon.

### WHAT WE ACTUALLY DID

Specifications were reconstructed from the source code and from production behaviour, rather than from interviews or from the existing documents. Where the code and the documents disagreed, the code was treated as authoritative — it is, after all, what has been running. Where the code and production behaviour disagreed, that discrepancy was itself a finding worth escalating.

### THE CHALLENGES WE ENCOUNTERED

- **Intent is not recoverable from source.** Code records what a system does, never why. A rule that looks arbitrary may encode a regulatory requirement, a decade-old incident, or a mistake nobody noticed — and the three are indistinguishable from the source alone.
- **Institutional memory had already left.** The people who could have supplied intent had retired. Those remaining offered recollections that, when tested against the code, were frequently and confidently wrong.
- **Volume.** An estate of this age contains an enormous quantity of behaviour, most of which is routine. Documenting everything at equal depth produces a document nobody reads, which is a different way of having no documentation.

### HOW WE RESOLVED THEM

- Documentation distinguished rigorously between observed behaviour and inferred intent. Behaviour was stated as fact; intent was labelled as inference and marked for confirmation. The two were never blended into a single confident sentence.
- Recollections were treated as hypotheses to be tested against the code, not as inputs to be recorded. Where memory and code diverged, the divergence was documented explicitly, because it is precisely the kind of gap that causes a future change to go wrong.
- Depth was allocated by risk. Money movement, regulatory calculation and customer-facing behaviour were documented exhaustively. Routine paths received proportionate treatment.
- Documentation was generated from the source and regenerated on change, so that it degrades far more slowly than a hand-maintained document — which begins drifting from reality on the day it is signed off.

### HOW WE PROVED IT WORKED

Reconstructed specifications were validated by using them: engineers with no prior exposure to the estate were asked to predict system behaviour from the documentation alone, and their predictions were tested against the running system. **Where a prediction failed, the documentation was wrong** — not the engineer — and it was corrected. This is a slower standard than review, and it is the only one that produces documentation which can actually be relied upon.

**RISK AVOIDED**

Key-person dependency surviving the migration intact. A modernised estate that only three people understand is a modernised estate that carries the original risk in a newer language — and the newness makes the risk harder to see, not easier.

**WHAT A US BANK SHOULD TAKE FROM THIS**

Documentation is not a deliverable produced at the end of a migration. It is the artefact that determines whether the migration reduced your risk or merely relocated it into a language your current staff happen to be more comfortable reading.

## WHAT A US BANK SHOULD TAKE FROM THIS

The Thai engagement is not the point. The point is that all four migrations — language, platform, database and architecture — were executed on one estate, under one evidentiary standard, without a big-bang cutover. Most vendors perform one of the four and describe it as modernisation.

| YOUR CONSTRAINT | HOW THIS ENGAGEMENT ADDRESSES IT | MAPS TO |
|---|---|---|
| **Model risk validation** | Deterministic generation combined with per-path parity evidence. Reproducible by construction, which is the precondition for validating anything at all. | SR 11-7 |
| **Operational resilience** | Incremental cutover with the system of record intact throughout, and rollback available at every phase. No single irreversible switch. | OCC OP RESILIENCE |
| **Undocumented estate** | Specifications reconstructed from source and validated by prediction, not from institutional memory that has already left the building. | KEY-PERSON RISK |
| **Licence cost** | Oracle and DB2 retired to PostgreSQL, with semantics and access paths reconciled row by row rather than assumed. | RUN-RATE |
| **Change velocity** | Domain services replace a single deploy unit, so change ceases to be an all-or-nothing event requiring institutional courage. | RISK-BASED SUPERVISION |
| **Audit defensibility** | Every migrated path has a parity artefact that an examiner can inspect independently of the delivery team's account of it. | INTERNAL AUDIT |

### 4 in 1

Language, platform, database and architecture — migrated on the same estate, in the same programme, under the same evidentiary standard. The engagement is bound by non-disclosure, so the client is not named and the metrics are not published. The method is what transfers, and the method is what we are offering to walk your team through in detail.

## THE QUESTIONS WORTH ASKING ANY VENDOR

- Is your code generation deterministic? If the same input produces different output on different runs, how does my model risk function validate it?
- What does your generated code do with packed-decimal rounding? Show me the evidence, not the approach.
- How do you prove behavioural equivalence? Show me the artefact an auditor would inspect.
- What is your rollback position at each phase? If the answer involves a single cutover weekend, what happens if parity fails on the Monday?
- How were the service boundaries derived? If the map resembles our org chart, where did it actually come from?
- How is documentation validated? Review by the team that wrote it is not validation.

## WHERE THIS GOES NEXT

> If you operate a mainframe, you already know the cost of leaving it alone, and you have very probably been quoted a modernisation that asks you to trust the result. We would rather show you how the evidence gets produced.

TECH4BIZ SOLUTIONS | CODENUK | AUDITTRAX LABS

tech4bizsolutions.com codenuk.com auditraxlabs.com

contact@tech4biz.io