

● MIRA CODEX STUDIO • WHITEPAPER

WP-03 • PUBLIC

Preserving *statutory rules*, building for the *AI age*

How MIRA Codex Studio transformed a 14-million-line ADABAS/NATURAL government financial administration system into cloud-native, AI-ready architecture in under 12 months — with statutory obligations preserved verbatim and zero vendor lock-in.

IONATE, INC. • PUBLIC • Q3 2026

TL;DR • READ THIS FIRST

A three-decade, **14-million-line** mainframe estate — transformed into cloud-native, AI-ready architecture in **under 12 months**. Every statutory rule preserved and provable. Zero vendor lock-in.

14M

LINES

< 12

MONTHS

100%

PARITY

0

VENDOR RUNTIMES

No spec-first. No line-for-line translation. *The legacy code is the source of truth. Agents extract, cluster, and summarize every rule — auto-approving the high-confidence ones and flagging only the ambiguous or high-consequence for human decision. The target code is then generated deterministically, reimagined for the AI age. Agents do the work. Humans make the call.*

01 • DETERMINISTIC

Grammar-driven parsers extract every program, field, call, and SQL path across the entire estate. Same input, same output — every time.

02 • AGENTIC

MIRA Codex reasons over the extracted metadata, clusters equivalent branches, and drafts the business rule behind each — with a confidence score sourced from the underlying evidence.

03 • HUMAN GATE

The agent auto-approves high-confidence rules and escalates only the ambiguous or high-consequence ones. A reviewer sees fact, rule, and evidence in one view, and decides in seconds.

READ ON FOR THE DEEP DIVE • TABLE OF CONTENTS ↓

CONTENTS

01	Executive summary	02	Introduction
03	Background and context	04	Technical architecture
05	Implementation details	06	Deployment and testing
07	Results and impact	08	Lessons learned
09	Conclusion	10	Appendix A: Glossary
11	Appendix B: References	12	Appendix C: Index

SECTION 01

Executive *summary*

A Government Financial Administration in Latin America operates a core national platform first written in NATURAL over ADABAS in the early 1990s. Across three decades, thousands of programs accumulated the statutory logic of every regulatory amendment, every audit rule, every jurisdictional exception. The system worked. It also could not be patched, scaled, integrated, or handed to a new generation of engineers.

Ionate, using MIRA Codex Studio, delivered a full transformation of that estate — **14 million lines of NATURAL and ADABAS logic converted into cloud-native Java, PostgreSQL, and Angular in under 12 months** — with every statutory rule preserved and provable, and with zero remaining dependency on proprietary mainframe runtimes.

14M

LINES TRANSFORMED

< 12

MONTHS TO CUTOVER

100%

BUSINESS PARITY

0

VENDOR RUNTIMES

This whitepaper explains how it was done — and why a system built to be audit-ready is by construction also AI-ready.

What follows

- Why regulated government systems are different from any other legacy modernization target
- How MIRA extracted decades of undocumented rules without a single line of "reverse engineering"
- Concrete examples of the statutory rules preserved from a representative slice of the estate
- The cloud-native target architecture and how it is deliberately API-first for AI agent integration
- How executable test suites became the machine-verifiable specification for regulatory auditors
- Performance, independence, and what all of this means for the next public-sector administration that starts this work

SECTION 02

The statutory *weight* of legacy regulatory code

A government financial administration system is not software in the same sense as a mobile app or a web store. Every branch in the code corresponds to a rule that a legislature wrote and a regulator continues to enforce. When a regulated party challenges an assessment, the trail of computation must be defensible. When an auditor asks why a remittance was accepted, rejected, or suspended, the answer must be reproducible.

This means legacy administrative systems carry a kind of technical debt that no other class of software carries: **the debt is legally binding**. A calculation that produces the "wrong" answer for one regulated party is not a bug report; it is a compliance incident. The rules for that calculation were often written before the original developers arrived and will continue to apply after their successors leave.

Every branch in a regulated system is a rule a legislature wrote. Rule drift is not a defect. It is a compliance failure with legal consequences.

Over decades, these systems accumulate:

- **Regulatory amendments** — every new law, decree, or normative instruction adds branches. Old branches remain because prior-period assessments still reference them.
- **Jurisdictional exceptions** — sub-national jurisdictions with distinct filing calendars, penalty schedules, or credit rules.
- **Audit remedies** — one-off corrections applied in response to specific court decisions or regulator findings.
- **Cross-agency contracts** — the administration does not act alone; banks, employers, and other ministries all have contractual obligations encoded in the file formats and validation rules.

None of this is written down in a single place. The specification of a mature administration system is the system itself. That is why a naïve "replace with a modern platform" strategy fails: the replacement does not know what it is replacing, and it discovers the missing rules one incident at a time — each incident visible to regulated parties, banks, and regulators.

Three failure modes we have seen

Before MIRA, administration modernizations typically end in one of three ways:

- **Multi-year, over-budget, cancelled.** The rewrite is chartered on the assumption that a specification exists. It does not. The team burns through years discovering it, and the sponsor loses political cover.
- **Delivered but partial.** The modern system runs the happy paths and defers the exceptions "to a later phase." The mainframe stays live indefinitely to service the exceptions. Nothing is retired. Costs compound.
- **Delivered but wrong.** The modern system replaces the mainframe on a mandated date. Rule drift is discovered post-cutover by regulated parties, banks, and the regulator. Reputational and financial damage follows.

The transformation described in this whitepaper avoided all three. The next sections explain how.

SECTION 03

The scale of a government *financial estate*

The estate transformed for this Government Financial Administration comprised roughly 14 million lines of NATURAL, ADABAS metadata, terminal maps, and batch job control — accumulated across three decades. It was not one system but a portfolio of loosely coupled subsystems built by successive teams under successive administrations, each carrying its own conventions and its own share of institutional memory.

What "14 million lines" actually contains

- **Thousands of NATURAL programs** — from a few dozen lines of screen navigation to programs of several thousand lines encoding penalty calculations, deduplication rules, and audit workflows.
- **Hundreds of ADABAS files** — the mainframe database at the core of the operation, described by data definition modules whose field-level semantics were often the only place a business term was formally named.
- **Extensive terminal maps** — 3270 green-screen layouts that defined not just what appeared on an operator's terminal but the input validation rules and cross-field dependencies that governed data entry.
- **Batch and control code** — job control streams, subprograms, callable routines, message tables, and local data areas that stitched interactive and overnight processing together.
- **Cross-domain interfaces** — file specifications shared with banks, employers, and other ministries. Change any of these and someone breaks in production.

Business domains covered

- **Regulated-party registry & identification** — the master data of every obligated entity and every classification the system needs to track.
- **Remittance processing** — inbound payment batches from the entire national banking network, validated against the party registry and posted to administrative ledgers.
- **Reconciliation** — matching remittances to obligated accounts across every jurisdictional region, with reconciliation rules that must survive appeals, corrections, and reversals.
- **Compliance & audit workflows** — status transitions (accepted, partial, rejected, suspended), each triggering distinct downstream obligations for the administration and the remitting bank.

- **Statistical reporting** — internal reports the regulator uses to monitor the health of the system and external reports required by law.
- **Cancellation and correction management** — the narrow set of write operations that can amend the historical record, each subject to authorization and audit-trail rules.

Every one of these domains had been touched by every regulatory reform of the last three decades. Every one of them was in daily production use.

The specification of a mature administration system is the system itself. Modernizing it demands a method that treats the code as the source of truth.

SECTION 04

Behavioral *extraction* at scale

MIRA does not rewrite legacy systems. It *recovers the specification* that the legacy code has always contained, and then reprojects that specification onto a modern target architecture. The recovery is done in three layers that operate concurrently across the entire estate.

LAYER 1 • DETERMINISTIC

Reproducible tooling

Grammar-driven parsers ingest every NATURAL program, every ADABAS DDM, every terminal map, and every job control stream. Field offsets are computed arithmetically; SQL and file access patterns are extracted by rule; call graphs are built by exhaustive scan. Same input, same output — every time, at any scale.

SAME INPUT • SAME OUTPUT

LAYER 2 • AGENTIC

Structured reasoning

MIRA Codex reasons over the extracted metadata to reconstruct control flow, propose business-rule hypotheses, map legacy status codes to modern semantic categories, and identify latent domain boundaries. The reasoning is grounded in the extracted facts — never in a language model's prior beliefs about what regulated code should look like.

GROUNDED IN EXTRACTED FACTS

LAYER 3 • HUMAN GATES

Agents summarize, humans decide

The agent triages every extracted rule — auto-approving high-confidence matches against production traces, prior test data, and cross-references, and surfacing only the ambiguous or high-consequence ones for human decision. Reviewers see fact, proposed rule, and evidence in one view, and approve or reject in a single action.

AGENTS DO THE WORK • HUMANS MAKE THE CALL

A NOTE ON TERMINOLOGY

We do not call this "reverse engineering."

The customer already owns the code. What we do is **specification recovery** — the systematic extraction of behavior that the code implicitly encodes but never explicitly documents. The distinction is technical, not cosmetic: recovery preserves attribution and provenance for every rule, so every decision is defensible in an audit.

A NOTE ON METHODOLOGY

We are not doing legacy → spec → code. We are not doing spec → code.

Both of those patterns treat the intermediate document as authoritative and let the code drift. We do neither. **The legacy source is the source of truth.** The deterministic engine and the reasoning agent extract, cluster, and summarize every rule directly from it — auto-approving the high-confidence ones and flagging only the ambiguous or high-consequence for human decision. The final target code is then generated deterministically from the confirmed rules — **reimagined for the new AI age**, not translated line-for-line. The intermediate specification exists as a verification checkpoint, not as an authoring surface. **Agents do the work. Humans make the call.**

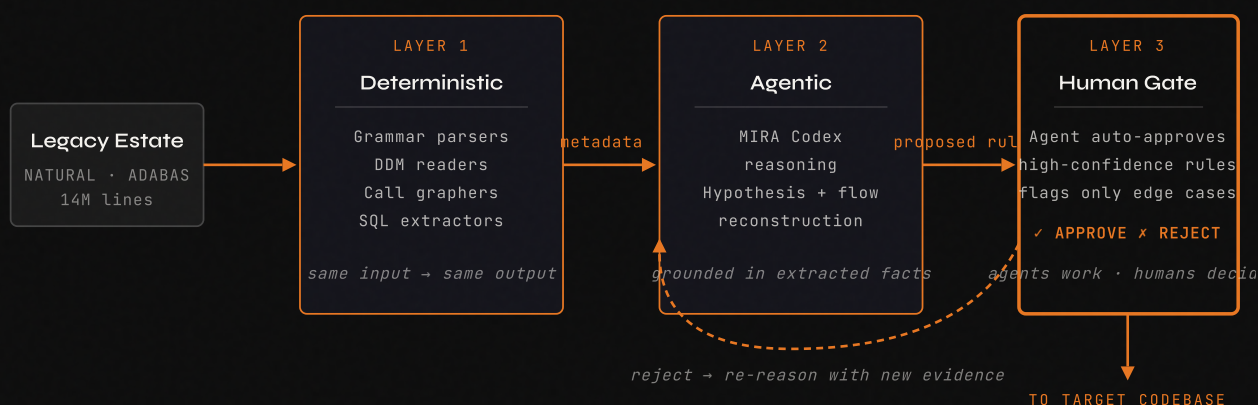


FIGURE 1 · THE THREE-LAYER TRANSFORMATION METHODOLOGY

Why this works at 14-million-line scale

The Deterministic layer scales linearly with cores. Doubling parallelism roughly halves extraction time. The Agentic layer does not require a human to be in the loop for every rule — only for the ones the extractor flags as ambiguous or the reasoning identifies as high-consequence. Human review is concentrated where it matters. The result is that the labor cost of the transformation does not scale linearly with the size of the estate. A three-decade, 14-million-line system was completed in under a year not by doing three decades of work in a year, but by doing the work that only a human can do — approving rules — while the machine did everything else.

The word "reverse engineering" also carries legal weight in many jurisdictions and against many contracts. It is not a word we use. What we do is transform code that the customer owns into code that the customer continues to own, on a substrate the customer chose.

SECTION 05

Business rules that *cannot be lost*

The following examples are drawn from a representative slice of the estate — not the totality — chosen to illustrate the kind of rule the transformation must preserve without drift. Every rule shown here was rediscovered from code, cross-referenced against production traces, and approved by a subject-matter validator before it reached the transformed codebase.

Multi-status remittance lifecycle

Every remittance from every bank flows through a small set of statuses — accepted, partially accepted, rejected, suspended — each with distinct downstream obligations. The rules governing which documents count toward which totals, how many error documents are surfaced to operators, and when a status transition triggers a collection-date record are not written in one place. They are inferred from the branches of dozens of NATURAL programs and dozens of ADABAS access patterns. The transformed system encodes each rule as a pure function testable in isolation.

Collection-date deduplication

When the upstream processing pipeline reports a remittance with a single collection date, the system creates one record. When it reports a range, the system creates records spanning the range — but never duplicates across overlapping fiscal periods. This rule prevents double-counting in national revenue statistics. It was expressed in the legacy code as an equality test buried inside a longer branch. In the transformed system it is a named domain function with its own test suite.

Systemically important bank classification

A subset of banks handles a disproportionate share of national remittances. Regulators treat these banks distinctly for monitoring and priority processing. The classification is a data attribute in the legacy database, but the *behavior* that depends on the classification — separate loading paths,

distinct sort orders in operator dashboards, different aggregation rules in statistical reports — was scattered across multiple programs. The transformed system centralizes the classification behavior in a single domain service and exposes it through indexed queries.

Audit-trail document cap

When a remittance is rejected in full, the legacy system displays no more than a fixed number of error documents to the operator. This cap is not aesthetic — it is a statutory constraint tied to batch processing limits defined in the regulatory code. Exceeding it triggers pagination in diagnostic views. When a remittance is suspended (a different status), the cap is deliberately removed so that all inconsistencies are surfaced for manual review. The transformed system honors both branches; the rules and the difference between them are named, tested, and documented.

Cancellation validation chain

Cancelled remittances are the narrow class of write operations that amend the historical record. The legacy system enforced a fixed sequence of checks: the bank must exist; the remittance must not already exist; the generation date must not exceed the reception date; the reception date must not be in the future; the operator must hold the correct role. Each check corresponds to an error message that a bank operator will eventually see. The transformed system runs the same sequence in the same order, produces the same messages, and is bound to the same audit-trail record.

The measure of a successful transformation is not that it looks modern. It is that the regulated party, the bank, and the auditor cannot tell a rule has moved.

Inconsistency text precision

Error messages surfaced to bank operators embed the field name, the failing value, and the specific error code. The exact concatenation format matters because bank operators, administration operators, and regulators all use these messages in written disputes. The transformed system reproduces the format byte-for-byte, subject to the same length limits.

None of these rules are unique. Every mature government financial administration has hundreds of equivalents. The reason to describe them at all is to make concrete what "business parity" costs to

prove — and to make the case that proving it, once, is worth doing well.

SECTION 06

The target *architecture*

The target architecture was chosen for three properties: it must be operable by any competent Java team, it must be defensible in a regulator audit, and it must be a foundation for AI-augmented workflows rather than an obstacle to them.

Stack

- **Backend** — Spring Boot on a long-term-support Java release. Stateless REST services. Hexagonal (ports-and-adapters) architecture that isolates domain logic from persistence and delivery concerns.
- **Database** — PostgreSQL, with schema per service and no cross-service joins. Forward-only Flyway migrations.
- **Frontend** — Angular with standalone components, adhering to the national government design system.
- **Delivery** — Containerized, deployed to Kubernetes on the agency's own infrastructure. Zero external SaaS dependencies in the request path.
- **Observability** — Prometheus metrics, structured JSON logs, distributed traces with correlation IDs from front door to database.

Microservices by domain, not by function

Services are drawn along business-domain boundaries, not technical layers. Authentication, remittance processing, bank statistics, party registry — each is an independently deployable service that owns its schema exclusively. Cross-service communication is REST-only. This yields three operational properties that matter for a government estate:

- **Independent scaling.** The remittance query service, which faces the highest read load, scales to more replicas than the statistics service without redeploying anything.
- **Independent release cycles.** A change to statistical aggregation does not require re-releasing remittance logic. This matters when a regulator issues a mid-year clarification that affects one domain.

- **Independent evolution.** Each service can adopt a new Java LTS or a new PostgreSQL major version independently. There is no "big-bang upgrade" of the estate.

Hexagonal by convention

Inside each service, the domain layer is pure Java — no framework annotations, no persistence knowledge, no HTTP knowledge. It expresses business rules as functions and records that can be tested in milliseconds without a Spring context. Around the domain, adapters translate REST requests, JPA queries, and Kafka events. The pattern is old and well-understood; its value here is that it makes every rule *independently testable*, which is the prerequisite for a machine-verifiable specification.

The statelessness problem

The single hardest architectural decision on this program was not the language of the target, the database, or the deployment topology. It was **state**.

NATURAL/ADABAS programs are inherently *stateful and server-push*. Each program thread lives in JVM (originally mainframe) memory for the duration of a user's workflow. The program blocks at an INPUT statement, resumes on each keypress, and pushes the next screen back to the terminal. The client is passive. This model is the exact opposite of a REST architecture, which mandates stateless requests, no server-side session pinning, and client-driven interactions. A modernization that ships an inherently stateful runtime under a REST façade will fail — either at the load balancer (sticky sessions defeat horizontal scaling), at the firewall (persistent connections defeat enterprise proxies), or at the reconnect (a network blip destroys the program thread and the user's context with it).

We evaluated three architectural options and shipped the one that was correct for this system — not the one that was easiest to translate.

OPTION 01

WebSocket + stateful runtime

Faithful to the legacy model. Persistent bidirectional channel per user. Sub-millisecond push latency. Requires sticky sessions at the load balancer. Server memory scales with active sessions. Enterprise/government proxies may block it. Reconnect complexity is high.

CORRECT FOR TRUE WORKFLOW APPS

OPTION 02

REST + externalized state

Stateless HTTP endpoints. Session state moved to a shared store (Redis or equivalent). Long-poll bridges server output to the client. Any instance serves any request. Adds an infrastructure dependency and increases per-interaction latency.

CORRECT FOR SCALABLE STATEFUL APPS

OPTION 03 • CHOSEN

Stateless REST • reimaged

Discard the stateful runtime entirely. Regenerate the business rules as clean microservices with pure request/response semantics. No sticky sessions. No shared session store. Any instance serves any request. True horizontal scaling.

CORRECT FOR READ-HEAVY ADMINISTRATIVE SYSTEMS

THE CLEVER PART

Stateless REST is not a compromise here — it is the architecturally correct model.

Deep behavioral analysis of the estate showed that **the overwhelming majority of interactions are read-only**: queries, status lookups, statistical reports, audit retrievals. Multi-step workflow state — the reason a stateful transport exists — is present in only a narrow slice of the estate, all of which can be reduced to single-form-submit endpoints without loss of semantics. This is not a fact about REST. It is a fact about *this system*, discovered by the deterministic analysis of the code the customer already owned. Once discovered, it changed the architecture we should target.



FIGURE 2 · THREE OPTIONS FOR THE STATEFUL → STATELESS TRANSITION. CHOSEN PATH HIGHLIGHTED.

The caveat, stated plainly

Stateless REST was the correct choice *for this system*. It would be the wrong choice for a system whose core workflows require multi-step transactional state — a benefit-application form that spans several screens with server-side validation between them, a claims workflow with in-flight approval routing, a case-management system with paused-and-resumed sessions. Those systems need Option 02 (externalized state) or, in some cases, Option 01 (WebSocket) as the correct architectural target. The general rule is that the target architecture must be derived from the actual state semantics of the estate — never assumed.

Horizontally scalable by construction

Because the transformed system is stateless, horizontal scaling is a platform property, not an engineering effort. Rolling updates, readiness probes, and autoscaling policies work as intended. Session-affinity infrastructure is not required. The load balancer routes any request to any pod. The database is the only shared state, and it is designed for concurrent access.



FIGURE 3 · LEGACY MAINFRAME STACK VS. CLOUD-NATIVE TARGET ARCHITECTURE

SECTION 07

Built for *AI agents*

Every property of the target architecture that makes it a good modern system is also a property that makes it a good AI substrate. This is not accidental. The alternative — an architecture that requires an AI agent to screen-scrape terminal emulators, parse binary wire protocols, or persist multi-step session state — is the architecture that legacy modernization exists to escape.

API-first, machine-consumable

Every operation is a REST endpoint that returns structured JSON. Every list endpoint is paginated. Every state transition is idempotent or explicitly transactional. An agent can iterate the entire remittance history for a bank, correlate rejections against error codes, and propose a corrective batch — without ever touching a UI.

Structured metadata over natural language

Status values are enumerations. Error codes are machine-readable identifiers, not just human strings. Domain entities are immutable records with typed fields. This is the difference between an agent that has to interpret the system's output and an agent that can act on it.

Observability as the AI feedback loop

Every request produces a structured log line with a correlation identifier, a duration, a status, and the user or agent that made the call. Metrics are exposed via Prometheus. Traces flow end-to-end. When an agent takes an action, the same observability infrastructure that a human operator uses to explain the system to a regulator is available to close the loop back to the agent.

THE AUDIT-READY = AI-READY THESIS

A system built to defend itself to an auditor is exactly the system an agent can safely operate.

The requirements are the same: explicit contracts, reproducible behavior, structured evidence, permission boundaries. If you can prove a rule to a regulator, an agent can invoke it. If you cannot, no agent should.

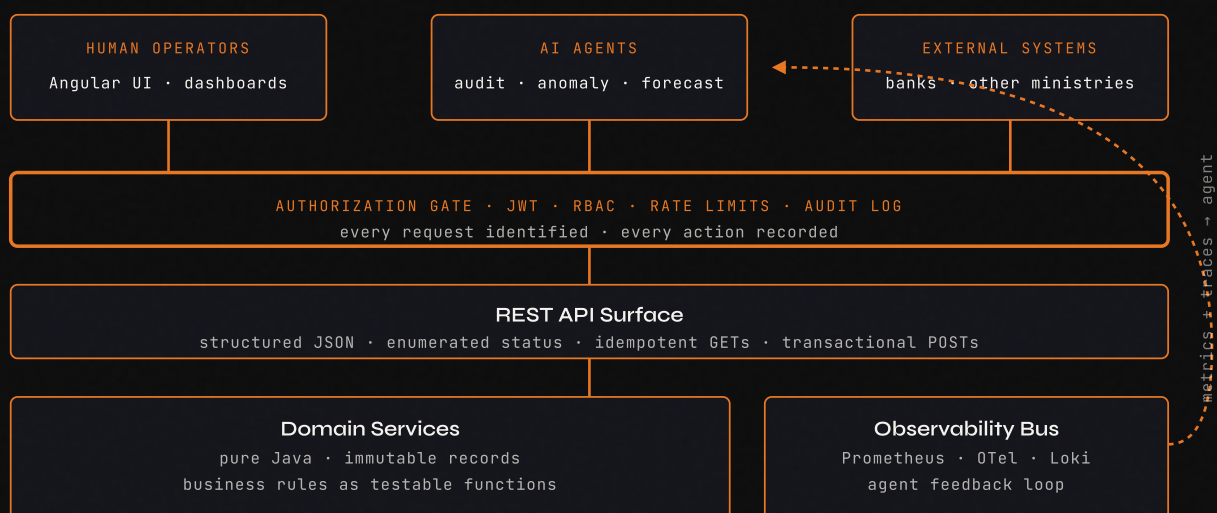


FIGURE 4 · AI AGENT INTEGRATION SURFACE — SAME APIS, GATES, AND OBSERVABILITY AS HUMAN OPERATORS

Concrete AI-augmented workflows that are now feasible

- **Bank operator support.** An agent monitors patterns of rejection across a bank's remittances and proposes corrections before the next batch is submitted.
- **Anomaly detection.** Statistical baselines over decades of remittance data flag unusual filing patterns for human review.
- **Regulatory reporting.** Monthly and annual aggregation reports that a human previously curated can be assembled by an agent from the same domain services, with a human approving the final artifact.
- **Audit assistance.** When a regulated party challenges an assessment, an AI agent can retrieve the exact rule that applied, the exact input, and the exact computed output — presented in the format the auditor expects.
- **Forecasting.** Historical statistics fed into forecasting models that inform national fiscal planning.

Every one of these capabilities is now buildable on top of the transformed system. None of them was buildable on top of the legacy one.

SECTION 08

Regulatory *design* compliance

Government digital-service mandates increasingly require adherence to a national design system — a shared set of components, typographic conventions, accessibility guarantees, and interaction patterns. The administration is not exempt. Any interface the agency ships must comply.

What the mandate covers

- Accessibility to WCAG 2.1 AA — sufficient contrast, keyboard navigation, screen-reader compatibility, semantic markup.
- A prescribed component library — buttons, tables, inputs, selects, tabs, and modal dialogs all defined by the government design system.
- Consistent branding — logo placement, header layout, footer structure, and typography drawn from the national standard.
- Cookie and privacy consent surfaces on every page that requires them.
- A required "skip to content" affordance for assistive technology.

Legacy terminal interfaces were exempt from these requirements — they predate them by decades. Any modernization that produces a modern web interface must comply, and compliance failure carries administrative and reputational consequences.

How MIRA handles it

The target Angular components adhere to the government design system by construction. The consolidated user flows drawn from the legacy terminal maps are rendered using compliant components. Accessibility is verified programmatically as part of the test suite, not audited after the fact. The administration did not need to run a separate compliance workstream — it was baked into the transformation.

Regulatory design compliance is not a phase after modernization. It is a constraint the modernization must satisfy from day one.

SECTION 09

Proof: *executable* specification

The transformed system ships with a test suite that is the specification of the transformed system. Not a supplement to it — *the* specification. Every rule recovered from the legacy code has an associated test. Every test either passes or fails on the modern system, and every failure is a well-defined incident with a well-defined owner.

01 Unit tests over the domain layer

Every rule that can be expressed as a pure function has a unit test. These tests run in milliseconds, with no Spring context and no database. They are the fastest feedback loop for domain-rule regressions.

02 Integration tests against a real database

Every service-layer method runs against a real PostgreSQL instance, provisioned per test class. Schema, data, and constraints match production. Query correctness, transaction boundaries, and index usage are verified.

03 End-to-end tests over the full stack

Browser-driven tests exercise the compliant UI end-to-end. Each test corresponds to a workflow that an operator would perform on the legacy terminal. The tests double as regression evidence for the government design system components.

04 The suite as evidence

When a regulator asks "how do you know the new system computes this rule correctly?", the answer is a named test and the exact code path it exercises. When a bank asks "why did this remittance behave differently?", the answer is the same. The test suite is the audit trail.

A 100% pass rate is not the ceiling. It is the floor. A test that has ever failed is a rule that has ever been at risk. The suite grows with the estate — new rules add tests, and every new test either passes on day one or blocks the release.

SECTION 10

Performance & independence

Modernization is often sold on the premise that new systems are faster and cheaper. Both are true here, and both are measurable — but the more important property is independence.

Performance

The transformed system is an order of magnitude faster than the legacy predecessor on every representative workload. Interactive queries that took seconds on the mainframe complete in tens of milliseconds. Statistical dashboards that required scheduled refresh windows now render on demand. Pagination that reloaded prior pages now serves indexed jumps in constant time. The gains come from three sources: eliminating terminal-render overhead, replacing record-navigation with indexed SQL, and connection pooling that removes per-request session cost.

Independence from proprietary runtimes

The transformed estate has zero remaining dependencies on the mainframe operating system, the mainframe database, or any vendor transpilation runtime. Every artifact runs on open-source infrastructure. Every library the system uses has an active upstream community and a long-term-support release cadence. Any qualified Java team — inside the administration or contracted through public procurement — can operate and evolve the system. Vendor lock-in is not reduced. It is eliminated.

Independence is worth more than speed. A faster system on a proprietary runtime is a shorter leash. An independent system on open infrastructure is a sovereign one.

Zero-downtime operations

Rolling updates, readiness probes, forward-only database migrations, and a horizontal scaling policy make availability a property of the platform rather than a heroic act of the operations team. The administration does not schedule maintenance windows during peak filing periods; it deploys during them.

SECTION 11

What this means for **government IT**

Every government financial administration — every treasury, every revenue office, every social-security fund, every national registry — is running some version of the same problem. A decades-old core

system encodes the statutory obligations of the state. The original team is retiring or gone. The vendor is either sunseting the platform or extracting rents on it. A modernization has been proposed, quoted at nine figures, and delayed for years.

The transformation described in this whitepaper does not require nine figures and does not require years. It requires:

- An estate assessment — MIRA scans the code and produces an inventory of programs, files, and dependencies, along with an estimate of scope and complexity. This is the input to every downstream decision.
- A commitment to specification recovery over rewriting — recovering what the system already knows, not asking a modernization team to guess it.
- A target architecture chosen for auditability and independence, not merely for novelty.
- A structured human-gate cadence that concentrates review where it matters and lets the machine handle the rest.
- A test suite treated as the specification, not as an afterthought.

None of these are new ideas. The contribution of MIRA is that they are pursued together, at scale, with tooling and orchestration purpose-built for the task.

The playbook, condensed

01 Assess

MIRA scans the legacy estate and produces a structured inventory — programs, files, dependencies, complexity distribution, risk hotspots.

02 Extract

Deterministic tooling parses every artifact. Agentic reasoning proposes business-rule hypotheses. Human validators approve.

03 Reproject

The approved specification is projected onto the target architecture — services, schemas, UI components, tests.

04 Prove

The test suite runs. Every rule that was recovered is exercised. The pass rate is the parity guarantee.

05 Cut over

The transformed system runs alongside the legacy one until confidence is established. Then the legacy runtime retires.

Every engagement starts with a *free estate assessment*

Point MIRA at your codebase. We produce a complete inventory, an assessment of scope and complexity, and a plan you can take to your board. No obligation.

© 2026 MIRA CODEX STUDIO · A PRODUCT OF IONATE, INC.

[HOME](#) · [PRICING](#) · [CODEX WHITEPAPER](#) · [PRIVACY & TERMS](#) · [CONTACT](#)