

● MIRA CODEX STUDIO • WHITEPAPER

WP-04 • PUBLIC

Deterministic engine. *Reasoning* agent. *Human* gate.

How MIRA Codex Studio combines reproducible tooling, agentic reasoning, and structured human approvals to transform legacy systems without drift — proven across an IBM CSP estate with vendor-binary gaps and an Oracle Forms enterprise retirement.

IONATE, INC. • PUBLIC • Q3 2026

TL;DR • READ THIS FIRST

One methodology. Two extreme cases. **Zero drift**. A vendor-binary mainframe estate with missing source, and a discontinued-platform enterprise retirement — both transformed on the same three-layer pattern.

3	2	100%	0
LAYERS	CASE STUDIES	BUSINESS PARITY	RULE DRIFT

Deterministic tooling extracts every rule reproducibly. **Agentic reasoning** clusters, summarizes, and scores each rule against its evidence. **Human gates** decide only on what the agent flags — never on what it already proved. **Agents do the work. Humans make the call.**

01 • DETERMINISTIC

Grammar-driven parsers, AST walkers, dependency graphers — same input, same output, at any scale. Structural facts about the estate become the ground truth.

02 • AGENTIC

MIRA Codex reasons over the extracted facts to reconstruct control flow, propose the business rule behind each branch, and attach a confidence score sourced from the evidence.

03 • HUMAN GATE

Agent auto-approves high-confidence rules; escalates only ambiguous or high-consequence ones. Reviewers see fact, rule, and evidence in one view — decide in seconds.

READ ON FOR THE DEEP DIVE • TABLE OF CONTENTS ↓

CONTENTS

01	Executive summary	02	Introduction
03	Background	04	Methodology
05	Case study 1	06	Case study 2
07	Case study 3	08	Conclusion
09	Appendix A	10	Appendix B
11	Appendix C	12	Index

SECTION 01

Executive *summary*

Most legacy modernization programs fail on the same two questions: how do we know we captured every rule, and how do we know the machine did not invent any? MIRA Codex Studio answers both with a single methodology that combines three layers — a deterministic engine that produces reproducible structural facts about the estate, a reasoning agent that turns facts into rule hypotheses, and a human gate that decides only what the agent cannot prove on its own.

This whitepaper documents that methodology and shows it working under two very different kinds of pressure. The first case is an IBM CSP mainframe estate at a Latin American Municipal Government IT Agency, where the vendor runtime included binary components whose source was not available — a situation that would defeat any single-layer approach. The second case is a Tier-1 Latin American Employee Benefits & Compensation Fund's Oracle Forms enterprise portfolio, where the platform itself

was end-of-life and the deadline was existential rather than optional. Different domains, different constraints, same three-layer pattern, same outcome.

Agents do the work. Humans make the call. That is the difference between a modernization that ships in months and one that ships in years — or never.

What follows

- Why pure large-model transformation is not a serious methodology for regulated code
- The Deterministic Layer — what it actually does, and why "same input, same output" is a promise not a marketing claim
- The Agentic Layer — how MIRA Codex reasons over extracted facts without hallucinating rules that do not exist
- Human Gates that scale — the triage pattern that keeps humans in the loop for what matters and out of the way for what doesn't
- Two case studies illustrating the pattern under different constraints
- The approval state machine that carries every rule from proposal to production
- Honest limits — when this approach is the wrong choice, and what to do instead

SECTION 02

Why pure *LLM transformation* fails

It is tempting to point a modern reasoning system at a legacy codebase and ask it to produce the modern equivalent. The demos are convincing. The production track record is not. There are three reasons this pattern collapses under real-world modernization scale.

Reasoning without ground truth is guessing

A reasoning model trained on public code has strong priors about what business logic *usually* looks like. But every real legacy estate is full of code that violates those priors — historical corrections, regulatory carve-outs, one-off patches for specific counterparties. When the model reasons about such code, it defaults to the more common pattern in its training distribution. That default is a rule that

was never in the customer's system. Undetectable if the model is also the reviewer. Catastrophic if the modern system ships with it.

Reproducibility is a compliance requirement, not a nice-to-have

Regulated systems demand that the same input produce the same output on every re-run. Large models are not deterministic by construction — same prompt today, subtly different output tomorrow. A modernization that cannot be re-derived from the source is not a modernization. It is a snapshot that begins drifting the moment it ships. Every subsequent revision inherits the drift.

Scale exposes hallucination

A demo of a hundred lines can be inspected by a human. An estate of ten million lines cannot. If ten thousand transformed programs contain one hallucination each, the customer discovers them post-cutover, one incident at a time — every incident visible to end users, banks, regulators, and the press.

The value of a reasoning model is not that it replaces the deterministic step. It is that it makes the deterministic step legible to a human — quickly and at scale.

None of this is an argument against reasoning models. It is an argument for using them where they add value and refusing to use them where they don't. The three-layer pattern that follows is exactly that division of labor.

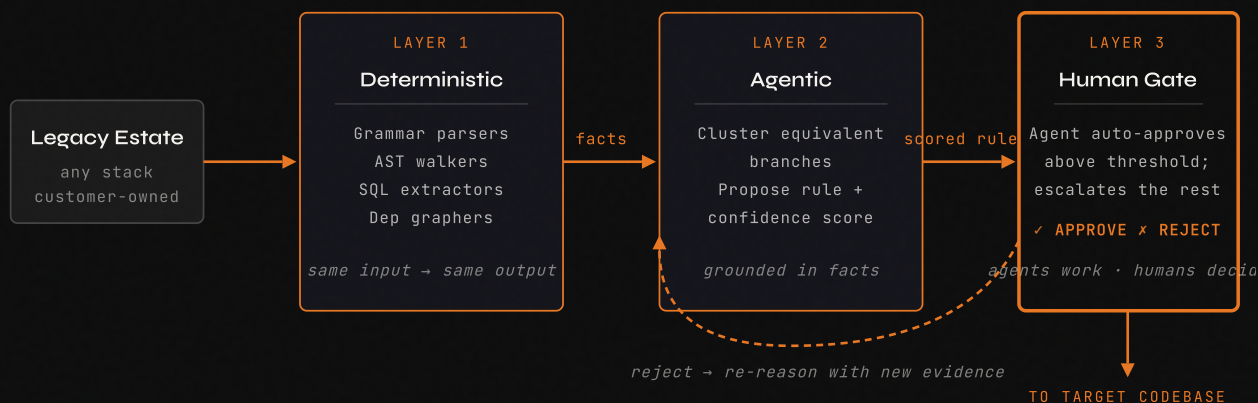


FIGURE 1 · THE THREE-LAYER METHODOLOGY

SECTION 03

The *deterministic* layer

The deterministic layer produces the ground truth on which everything else depends. Its job is to answer purely structural questions about the estate — what programs exist, what fields they read and write, what SQL statements they execute, what other programs they call, what screens they render, what error codes they surface. The answers must be reproducible: same source in, same fact set out, on every run, on any machine, at any scale.

What it actually does

- **Grammar-driven parsers** — one per source language (NATURAL, COBOL, RPG, Oracle PL/SQL, VB6, and so on) — turn source text into abstract syntax trees. No inference, no guessing.
- **AST walkers** extract every field definition, every arithmetic expression, every conditional branch, every loop, every I/O verb.
- **SQL extractors** pull embedded database statements and normalize them for downstream cross-reference.
- **Dependency graphers** record who calls whom, who reads what table, who writes what field.
- **Schema readers** ingest database and file definitions (ADABAS DDMs, DB2 catalogs, VSAM copybooks, Oracle DDL) and normalize them into a common relational model.

- **Message table parsers** extract user-facing text and error codes so they can be preserved verbatim.

Why "same input, same output" matters

Because a regulator, an auditor, or a customer inspector must be able to re-derive the same facts tomorrow that we derived today. Because when a transformed rule is disputed six months into production, the trace back to the original source has to reproduce byte-identically. Because if the ground truth can drift, everything built on it drifts too.

WHAT THIS LAYER IS NOT

The deterministic layer does not decide what code means.

It records what the code does structurally. Whether an if-branch encodes a statutory rule, a historical patch, or dead code is a question for the reasoning layer — grounded in the facts this layer produces, but not answered here.

Scaling behavior

Deterministic extraction is embarrassingly parallel. Doubling cores roughly halves wall-clock time until I/O becomes the bottleneck. This is the layer that lets MIRA process millions of lines in hours rather than months, without any degradation of correctness. It is also the reason the modernization does not become a linear-labor scaling problem: the extractor does not tire, does not miss lines, and does not skip programs because "they looked similar to something we already saw."

SECTION 04

The *agentic* layer

The agentic layer takes the structural facts produced by Layer 1 and asks the questions that only reasoning can answer. What does this branch *mean*? What business rule is this loop implementing? How do these three programs, each accessing the same table with slightly different filters, decompose into a coherent domain model?

Reasoning grounded in facts, not priors

MIRA Codex is instructed to reason only over the facts the deterministic layer produced — never over what "typical business logic looks like" from its training. This is enforced at the prompt level and validated at output time: every proposed rule must cite the specific facts (program name, line range, dependency edge, SQL statement, error code) that support it. A proposed rule with no citation is not a rule; it is a hallucination and is rejected before it ever reaches a human.

What it produces

- **Rule hypotheses** — natural-language statements of business rules, each attached to the evidence in the deterministic fact base that supports them.
- **Cluster maps** — equivalent or near-equivalent branches across multiple programs grouped so that one rule can cover them all.
- **Confidence scores** — every hypothesis carries a score derived from evidence quality: how many independent facts support it, how consistent they are, how well the rule matches observed production behavior in trace data.
- **Control-flow reconstructions** — decision trees showing how legacy inputs produced legacy outputs, so a modern equivalent can be tested against the same inputs.
- **Semantic mappings** — legacy status codes and enumerations mapped to modern semantic categories with the mapping justified by the evidence.

Where reasoning helps and where it doesn't

Reasoning helps where structure alone is not enough. Two programs that write to the same table with different filters usually indicate two aspects of a single domain rule — but only reasoning can distinguish "this is the base rule, that is the exception" from "these are unrelated concerns that coincidentally share a table." Reasoning does not help where structure is decisive — a SELECT statement's WHERE clause is what it is, and no reasoning layer is allowed to reinterpret it.

The agent is not the source of truth. The code is. The agent is the interpreter that makes the code legible — with citations back to the source for every claim.

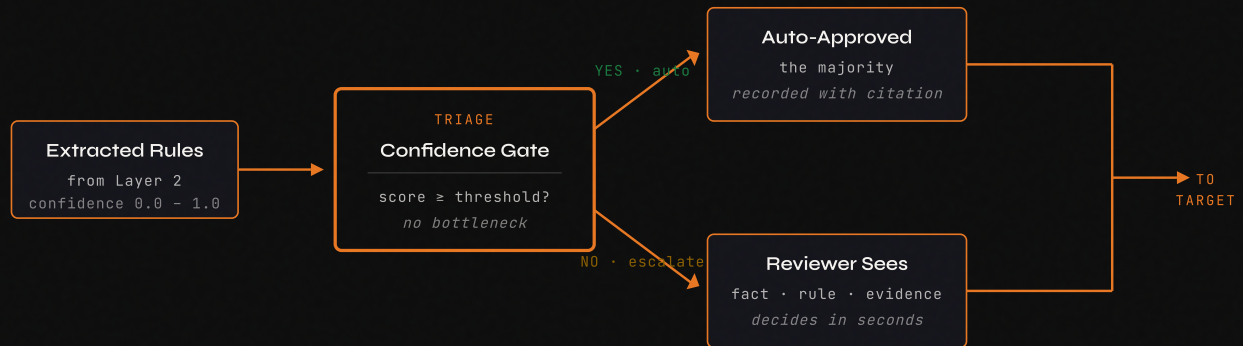


FIGURE 2 · CONFIDENCE-BASED TRIAGE — AGENTS AUTO-APPROVE THE MAJORITY, HUMANS DECIDE THE EDGE CASES

SECTION 05

Human gates that *scale*

The most common failure of "human-in-the-loop" is that the human becomes the loop. Every rule waits for a reviewer. The reviewer is the bottleneck. The modernization slows to the pace of the review queue. Weeks become months become years. The system never ships.

MIRA's human-gate design starts from the opposite premise. The reviewer's time is scarce and valuable. Every minute they spend confirming something the machine could have proven is a minute they didn't spend on something that only they can decide. So the triage is aggressive.

What the agent auto-approves

- Rules whose confidence score exceeds a project-configured threshold — typically because multiple independent facts support them, they match observed production traces, and no equivalent-but-conflicting rule was found elsewhere in the estate.
- Rules that were previously approved (with the same evidence) and are being re-derived on a subsequent run — regression paths reuse prior decisions.
- Rules whose transformation target is trivially structural — a field rename, a data-type mapping, an index carried forward — where there is no semantic choice to make.

What the agent escalates to a human

- Rules with low confidence scores — insufficient evidence, contradictory citations, or novel patterns not seen elsewhere in the estate.
- Rules the agent identifies as high-consequence — anything touching payment amounts, regulatory status transitions, cross-jurisdictional exceptions, cancellation or reversal paths, or authorization checks.
- Rules that conflict with a previously approved rule — the customer needs to decide which one is authoritative.
- Rules that reference behavior extracted from missing-source components (as in Case Study 1 below) — even a high-confidence reconstruction is worth a human check.

THE REVIEWER EXPERIENCE

One screen. Three panels. Two buttons.

Fact panel: the specific source lines the extractor pulled. Rule panel: the agent's proposed rule with its confidence score and reasoning. Evidence panel: production traces, test data, prior similar decisions. Approve. Reject. Both actions are recorded with the reviewer's identity and timestamp. Rejection routes back to the agent with the reviewer's note attached; the agent re-reasons with the new context.

The load calculation

For a well-structured estate, the auto-approval rate typically exceeds 90%. That means an estate of, say, 100,000 extracted rules produces on the order of 5,000 – 10,000 that a human ever sees. A reviewer working at ordinary attention pace clears each in under a minute. The math works. This is why "months" is a realistic timeline for estates that would take teams of engineers years by hand.

Agents do the work. Humans make the call. That framing is not marketing. It is the load model that makes the timeline defensible.

SECTION 06

Case study 1 • *vendor binary gaps*

MUNICIPAL GOVERNMENT • LATIN AMERICA • IBM CSP

An IBM CSP mainframe estate with ~30% of the runtime in vendor binaries and no source available.

A Latin American Municipal Government IT Agency runs its core administrative services on IBM CSP (Cross System Product) — a decades-old mainframe framework that combines terminal maps, COBOL program logic, DB2 access, and CICS runtime controls into a tightly integrated stack. The customer owns the source for everything they wrote. They do not own the source for portions of the vendor-supplied runtime, which is delivered as binary components.

Why this defeats a single-layer approach

A pure deterministic tool can parse everything for which source exists — but the missing runtime components hold critical execution semantics. A pure reasoning approach can hypothesize what the missing components do — but with no ground truth to anchor to, the hypotheses are unverifiable. The combination is where the value emerges.

How the three layers filled the gap

- **Deterministic extraction** processed all customer-owned source and produced the complete fact set for what the customer wrote. It also identified every place where customer code invokes a vendor runtime component and captured what the customer code expects that call to do (inputs passed, outputs consumed, error paths handled).
- **Agentic reasoning** combined those call-site expectations with several other signals: transaction traces from production, DB2 access patterns before and after each call, screen-capture archives from operator sessions, and the message-code catalog that the runtime emits on error. From these it reconstructed the behavioral contract of each missing runtime component — the pre-conditions, post-conditions, side effects, and error semantics — as a rule set that could be re-implemented in the modern target.
- **Human gates** reviewed the reconstructed contracts. Because these came from signal reconstruction rather than direct source, every one was escalated regardless of confidence score. Reviewers confirmed each contract against production traces, then approved. The re-implementation was then generated deterministically from the confirmed contracts.

The result

A modern Spring Boot service stack that behaves identically to the legacy system — including the parts whose original source the customer never had. Every business rule is now visible, testable, and revisable. The vendor binaries are retired. So is the license that came with them.

Missing source is not a wall. It is a signal-reconstruction problem — the same three layers, weighted toward the human gate because the evidence is indirect.

SECTION 07

Case study 2 • *discontinued platform*

EMPLOYEE BENEFITS FUND • LATIN AMERICA • ORACLE FORMS

A Tier-1 Latin American Employee Benefits & Compensation Fund retiring its Oracle Forms enterprise portfolio ahead of end-of-life.

The Fund's core operations — benefits enrollment, credit origination, collections, and member services — ran on Oracle Forms, a platform whose vendor deprecated new development in 2009 and ceased new releases in 2014. By the time the modernization began, the Forms runtime had no active security patches, could not be deployed to containerized infrastructure, and was on a countdown to complete environmental failure. The deadline was not "when we get around to it." It was existential.

What made this case different from Case 1

Source was available for everything. The problem was not missing information; it was that the source encoded rules in a form that could not survive the platform's death. Every trigger, every LOV binding,

every canvas layout, every PL/SQL block was tied to the Forms runtime — and the runtime was going away.

How the three layers applied

- **Deterministic extraction** parsed every form, trigger, block, LOV, and PL/SQL library into structural facts. Field bindings, database access patterns, canvas geometry, and message references were all captured as ground truth.
- **Agentic reasoning** reconstructed the business rules embedded in the trigger sequences — a class of code notoriously hard to read because logic is scattered across dozens of trigger types firing in a specific order. The agent identified which triggers formed coherent rules, clustered equivalent patterns across forms, and mapped legacy status codes to modern semantic categories.
- **Human gates** reviewed the escalated rules. Because the customer's operators had deep tacit knowledge of Forms behavior that was never written down, the reviewer role was staffed by domain experts as well as engineers. Every proposed rule was decided in seconds; the entire estate went through the queue at a defensible pace.

The result

The Forms portfolio is retired. In its place is a Spring Boot + Angular stack, containerized, observable, horizontally scalable, and independent of Oracle Forms in every meaningful way. Business rules that had lived in trigger-firing sequences are now named, tested, and documented. The Fund's operations continue without a visible change to members — and without the existential deadline that motivated the project.

*For the Oracle Forms transformation pattern in depth — including the SQL migration cheat sheet, the LOV-to-REST modernization, and the security and observability uplift — see the companion whitepaper **Oracle Forms to Next-Generation Applications**.*

SECTION 08

The approval *state machine*

Every rule that flows through the three layers moves through an explicit state machine. This is not documentation of the process — it *is* the process, encoded as a workflow that the platform enforces.

01 DRAFT

The agent has generated a rule hypothesis with a confidence score and citations. Not yet visible to anyone downstream.

02 PENDING_REVIEW

Confidence below threshold or flagged high-consequence. Queued for a human reviewer. Fact, rule, and evidence assembled into a single view.

03 APPROVED

Either auto-approved by the confidence gate or approved by a human reviewer. Attached to the rule are the reviewer identity (or "auto"), timestamp, and the exact evidence presented at the moment of decision.

04 REJECTED · RE-REASONING

Reviewer's rejection note is passed back to the agent with the original evidence. The agent generates a new hypothesis and returns the rule to PENDING_REVIEW. All prior decisions retained in the audit trail.

05 GENERATED

Approved rule has been rendered into modern target code by the deterministic generator. Test cases derived from the rule are attached to the generated artifact.

06 VERIFIED

Test cases pass on the generated code. The rule is now part of the shipped system. Any subsequent change to the rule re-enters the state machine at DRAFT with a link back to the prior version.

The state machine is not project management overhead. It is the audit trail. Every rule that ships in the modern system can be traced back to the exact source lines it came from, the exact reasoning that interpreted them, the exact reviewer who approved them, and the exact test cases that verified them. This is what "provable business parity" means in practice.

SECTION 09

Auditability by *design*

Regulated modernization stands or falls on whether the resulting system can be defended. Every rule must be sourceable to the legacy code that produced it. Every decision must be attributable to the person or process that made it. Every subsequent revision must be diff-able against the version it replaced. MIRA's three-layer methodology delivers this by construction, not as a follow-on documentation effort.

What is recorded, always

- **Source citation** — for every rule, the specific legacy source lines that produced it, referenced by program, file, and line range.
- **Reasoning trace** — the agent's proposed rule with its evidence and confidence score, retained even after human decision.
- **Decision identity** — the reviewer name (or "auto") for every approval, with timestamp and any note attached.
- **Evidence snapshot** — the production traces, test data, and prior decisions that were shown to the reviewer at the moment of decision, preserved so the decision context can be reconstructed later.
- **Generated artifact** — the target-code file, method, and test case that resulted from the approved rule.
- **Revision history** — every subsequent change to the rule linked back to prior versions.

What this enables

A regulator can ask why a specific transaction was processed the way it was. The trace goes: transaction → target code → generated rule → reviewer decision → agent reasoning → legacy source lines. Every step is recorded. Nothing is inferred after the fact. A subject-matter expert can be brought in to review a decision made months earlier and see exactly what evidence was presented at the time.

The audit trail is not built after the fact. It is the byproduct of the methodology running its normal course.

SECTION 10

When *not* to use this

An honest methodology is one whose limits are stated up front. The three-layer pattern is powerful for legacy transformation of code that the customer owns. It is not the right choice for several adjacent problems.

Green-field development

If there is no legacy code, there is no source of truth for the deterministic layer to extract from. Green-field builds should use appropriate design and development methodologies — the three-layer pattern has nothing to add.

Code the customer does not own

Third-party libraries, open-source dependencies, and vendor SDKs remain what they are. MIRA does not decompose them. The methodology addresses the customer's own estate; upstream code is treated as an opaque dependency that the modern target consumes on its own terms.

Estates where the customer intends to change the rules

The methodology preserves what the code does today. If the modernization program also intends to change business rules — deprecating a product line, restructuring an approval workflow, complying with a new regulation — those changes must be handled as a separate, deliberate design step. The three layers extract the current rules honestly; whether those rules should continue is a decision the methodology does not make.

Systems where speed matters more than correctness

Some transformations are throwaway — a quick proof of concept, a one-off report generator, an internal tool that will be replaced next quarter. The rigor of the three-layer pattern is not free. If the target system's cost of being wrong is low, faster and less careful approaches may be appropriate.

THE HONEST FRAMING

This methodology exists because the alternatives fail on regulated, mission-critical estates.

It is not the fastest way to transform any code. It is the fastest way to transform code that has to be right — and provably right — the first time it ships.

SECTION 11

Playbook for *adoption*

Adopting this methodology on a real estate is not a leap. It is a sequence of steps, each of which produces a defensible artifact.

01 Estate assessment

MIRA scans the legacy source and produces a complete inventory — programs, files, dependencies, complexity distribution, integration points, missing-source markers. This is a fixed-scope engagement with a defined output. It is the input to every subsequent decision.

02 Target architecture selection

Based on the assessment, select the modern target stack and — critically — the state semantics that fit the estate's actual workload. Do not assume; derive.

03 Deterministic extraction

Run the parsers, walkers, and graphers across the full estate. The output is the ground-truth fact base on which everything else depends.

04 Agentic rule generation

MIRA Codex reasons over the fact base and produces the initial rule set, each rule attached to its evidence and confidence score.

05 Triaged human review

Auto-approvals flow through immediately. Escalations queue for reviewers, who work through the backlog at pace. The reviewer pool is sized to the escalation volume, not the total rule count.

06 Deterministic generation

Approved rules feed the code generator. Target-language services, schemas, UI components, and test cases are emitted in a repeatable pass.

07 Test-driven verification

The generated test suite runs. Every rule that was approved is exercised. The pass rate is the parity guarantee to the customer and to the regulator.

08 Cutover

The transformed system runs alongside the legacy one until confidence is established. Then the legacy retires. The audit trail retires with it — preserved for the regulatory retention period, then archived.

Every engagement starts with a *free estate assessment*

Point MIRA at your codebase. We produce a complete inventory and an honest assessment of scope, complexity, and risk. No obligation.

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

[HOME](#) · [PRICING](#) · [WP-03](#) · [PRIVACY & TERMS](#) · [CONTACT](#)