thinkia

BROWNFIELD AI · WHITEPAPER · JUNE 2026

Legacy isn't rewritten. It's *decoded.*

Legacy code is decoded into specs and regenerated. The spec is the asset; the code, the by-product. A method for migrating critical systems with architectural intent.

EXECUTIVE SUMMARY

Every organization with history carries a system that holds the business together and that almost no one dares to touch. The logic lives in code few people understand, written by people who are no longer around. The option that looks prudent, rewriting it from scratch, is the most expensive and the one that loses the most rules along the way.

There is another reading. The value of a legacy system is not its code: it is the business intent trapped inside it. AI-assisted engineering can extract that intent, fix it in a governable specification, and regenerate the code against a target architecture. The spec becomes the durable asset. The code goes back to being what it always should have been: a consequence.

This document sets out the real problem with legacy, why the classic approaches fall short, and how a governed brownfield migration runs: the five-phase method, the known risks, and the four metrics that tell you whether it is working. It is not a product catalogue. It is a way of working.

37–42%

LESS EFFORT

in development versus the traditional method, observed across Coding 3.0 programs in production.[1]

75%

KNOWLEDGE WORKERS

already use AI at work, often without IT oversight.[2]

5

PHASES

from diagnosis to industrialization, with a quality gate at each one.

CONTENTS

01 The legacy problem p. 05
02 Why the classic approaches fail p. 06
03 The spec as the asset p. 08
04 Anatomy of reverse engineering p. 10
05 The method in five phases p. 11
06 Two real cases p. 13
07 Risk map p. 14
08 Success KPIs p. 15
09 Conclusion and manifesto p. 16

From the author

What migrates isn't the code. It's the *knowledge*.

For years I have discussed migrations of critical systems with technology directors, and the conversation almost always starts in the same place: how much code has to be rewritten. It is the wrong question. The code was never the problem. The problem is that half the rules that hold the business together are written down nowhere: they live in the heads of a few people and in conditionals no one remembers adding.

When we started working with AI agents in earnest, that question flipped. We stopped caring how much code the machine produced and started caring how well it recovered intent. A model that reproduces the syntax of a legacy system is of little use; one that recovers the rule that syntax hides changes everything.

That is where the thesis of this document comes from, and I will say it plainly. Legacy code is not rewritten; it is decoded into specs and regenerated. The spec is the asset. The code is the by-product. Once that idea is taken seriously, a migration stops being an act of faith: it becomes a process with traceability, quality gates, and human judgment.

It is not about writing more code. It is a different craft. The engineer stops typing and goes back to what they truly know how to do: deciding which rules matter, which architecture holds them, and which test proves they have been preserved. The machine generates; the person judges.

Anyone looking for a manifesto will find one at the end. Anyone looking for a method to move real systems without breaking what took years to build well will find that too. Both belong here, in that order.

David Alejano
Chief Strategy Officer · Thinkia · Madrid, June 2026
linkedin.com/in/dalejano

01

PART I

The problem and the *thesis*.

Why legacy code holds organizations back, why the classic approaches do not solve it, and what changes when the spec becomes the asset.

01

The system that holds the business together is the one no one dares to *touch*.

Legacy is not a technical problem. It is a dependency. A banking core, a policy system, a billing engine: decades of accumulated rules that work, that move real money, and that the organization understands less and less. The risk is not that the system fails. It is that no one knows any more why it works.

Four symptoms almost always appear together. The first is **technical debt**: every urgent patch left a scar, and the cost of maintenance grows while the capacity for change sinks. The second is dependence on **key experts**. Two or three people know the implicit rules; when they retire or leave, the knowledge goes with them and no copy remains.

The third is the fear of touching it. A one-line change can break something three modules away, and since no one holds the full map, the organization chooses not to move. The consequence is the fourth symptom: release cycles stretch out. What should ship in a week ships in a quarter, because every deployment requires a round of manual checking that stands in for the tests that were never written.

In regulated sectors there is a fifth. When a system is not traceable, the organization cannot show why it made a decision: why it denied a loan, why it settled a claim a certain way. Regulatory risk stops being hypothetical and becomes a fine with a date on it.

And meanwhile, teams are already generating code with AI on top of these systems, almost always without architectural context and without governance. AI speeds things up; but speeding up on top of a base no one understands only accumulates debt faster.

75 %

of knowledge workers already use AI at work, often without IT oversight.[2]

2–3

people typically hold the implicit business rules of an undocumented core system.

×4

typical gap between a healthy release cycle and one slowed by the fear of touching the legacy.

02

Each classic approach solves *half* the problem.

The four usual strategies for migrating legacy share one flaw: they treat the code as the asset and the business logic as a side effect. That is why each one recovers a part and leaves out the part that is hardest to rebuild.

APPROACH	WHAT IT PROMISES	WHY IT FALLS SHORT
Rewrite from scratch	*A clean, modern system, free of debt*	Rewrites the syntax but loses the implicit rules no one documented. A long, expensive, high-risk project: the new system is born not knowing what the old one did.
Incremental refactor	*Continuous improvement, controlled risk*	Keeps the original architecture and, with it, its limits. Improves the form without recovering the intent; debt is reduced, not cleared.
Lift-and-shift	*Move to cloud fast and cheap*	Changes where the system runs, not what the organization understands about it. The legacy stays a black box, now with a cloud bill.
API encapsulation	*Modernize access without touching the core*	Exposes the system without decoding it. Useful as a bridge, but it defers the problem: the critical logic stays locked away and without an owner.

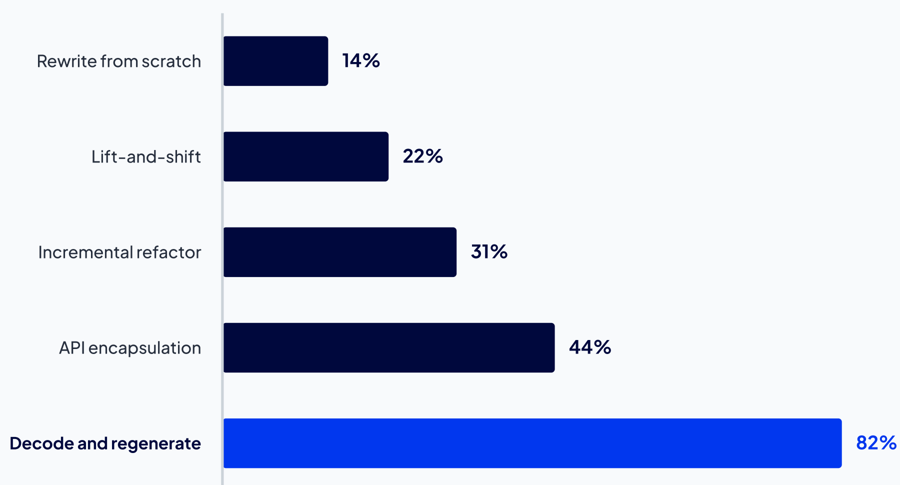
None of them is a mistake: each answers a partial question. The problem appears when they are chosen as a complete strategy, because then the implicit business rule, the most valuable and most fragile asset, goes unrecovered.

If you measure what really matters (how many business rules survive the migration), the order of the approaches inverts relative to their popularity. Decode-and-regenerate is the only one that recovers the near-complete set, because it is the only one designed to extract the intent before touching the code.

EXHIBIT 01

Only decode-and-regenerate recovers the complete set of rules.

Business rules recovered by approach, % (illustrative model)



Source: illustrative model · Thinkia analysis of real migration programs.

03

The spec is the asset. The code is the *by-product*.

For fifty years the code was the truth and the documentation was an approximation that aged worse than the system itself. AI-assisted engineering inverts that relationship. When a specification is precise enough, a model can generate the code that implements it, in whatever language and architecture is needed. What is scarce is no longer typing; what is scarce is clear attention.

A **spec**, in this sense, is not a requirements document that precedes development and is then abandoned. It is the contract the code implements, living and versioned: what the system does, which rules it respects, what it must never do. The code generated from it is disposable by design. If the target architecture changes, it is regenerated. If a better language appears, it is regenerated. The spec remains.

This reorders what is valuable in a migration. The asset the organization wants to protect is not the fifteen-year-old .NET or the monolith no one touches: it is the business rule that lives inside. Decoding it into a spec takes it out of the heads of three people and turns it into company property, readable by humans and executable by agents.

Human judgment does not disappear; it concentrates. Instead of being spread across thousands of lines, it is invested where the outcome is decided: which rules are correct, which architecture holds them, and which test proves they have been preserved. The machine takes on the syntax. The person takes on the judgment.

The spec is the *asset*. The code, the *by-product*.

04

Decoding recovers intent, not the *syntax*.

AI-assisted reverse engineering is not reading code and translating it. It is reconstructing, in three moves, what the system does and why. Each move produces an output a person reviews before going on to the next.

01

Extracting the business logic

The AI walks the legacy system and separates intent from implementation: which rules apply, in what order, with what exceptions. The implicit rules (the ones living in uncommented conditionals or in an expert's head) are made explicit and put on the table for validation.

02

Generating specs from the existing code

The recovered logic is fixed in a governable specification, with embedded documentation and traceability back to the source fragment. The spec stops depending on the old system: it is a standalone, versioned artifact that any AI-coding tool can consume to generate the new code.

03

Validation against the target architecture

Before a single line is generated, the spec is checked against the target architecture: what fits, what needs adapting, which rule clashes with the new model. The migration is validated against the target design, not against the system being left behind.

02

PART II

The method and the *execution*.

How a governed brownfield migration runs: the phases, the economics of effort, the cases, the risks, and the metrics that decide whether it works.

05

Five phases, a quality gate at *each one*.

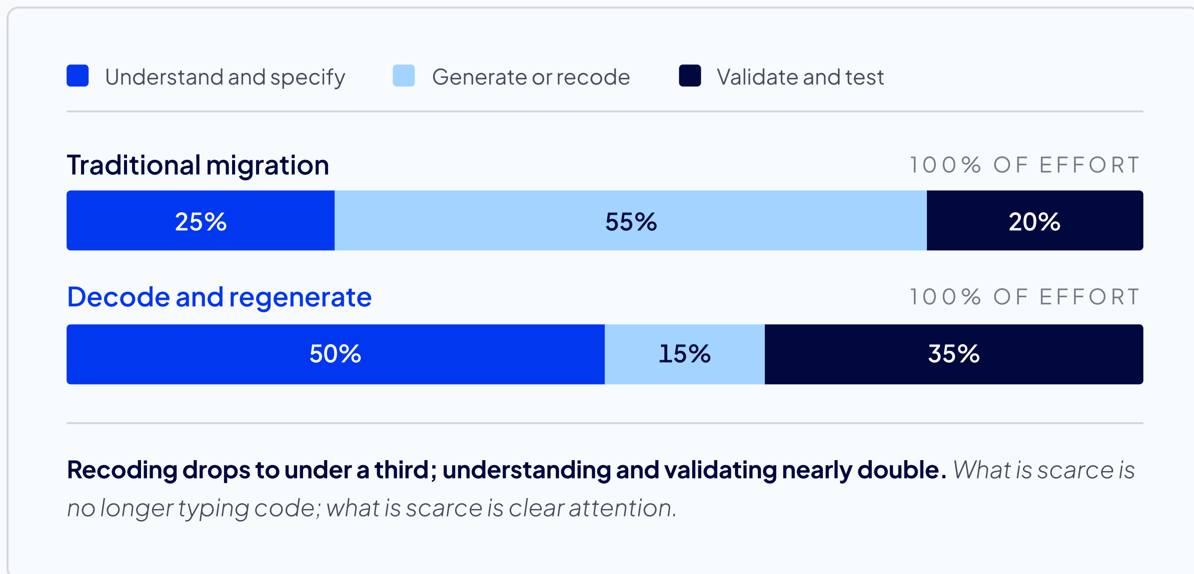
A governed migration moves through five phases. The human weight is greatest at the start, when you decide which rules matter and which architecture holds them, and the AI takes on more of the load as judgment gets fixed in specs. Diagnosis precedes commitment: the first phase delivers an actionable diagnosis before the organization ties itself to anything.

CROSS-CUTTING TRACK Architecture, business rules, and traceability · active in every phase

Order matters more than speed. **F0** diagnoses and qualifies the case; **F1** lays the foundations (target architecture, critical rules, first specs); **F2** tests the method on a lighthouse case with a business metric; **F3** scales to the adjacent domains; **F4** industrializes the practice so the organization can sustain it without Thinkia. No phase passes the gate without validating the business rules against the target design.

Where the effort goes when the spec is the asset

The method does not cut work by splitting the same load between human and machine. It redistributes it. In a traditional migration, most of the effort goes into recoding by hand. When you decode and regenerate, that block collapses and the one that truly protects the business grows: understanding, specifying, and validating.



Recoding drops to under a third; understanding and validating nearly double. *What is scarce is no longer typing code; what is scarce is clear attention.*

The net effect, observed across Coding 3.0 programs in production, is between 37 % and 42 % less development effort while holding quality.[1] It is not magic: it is no longer paying twice for the same knowledge, writing it once in the spec and regenerating the code as many times as needed.

06

Two migrations, one *discipline*.

The cases are anonymized by confidentiality policy. They matter for what they teach about the method, not as sales proof: where the friction was and what was learned.

CASES IN PRODUCTION

The implicit rule is always the *hard work.*

Private banking

A brownfield migration of a .NET core inside a phased program, with diagnosis before commitment and sponsorship at the highest level. Reverse engineering surfaced rules no document captured; the human validation gate stopped architecture drift before it reached production.

Global insurer

A two-phase Coding 3.0 program. Definition (a pipeline of specification agents, technical design, and REEF CORE across four pilot projects) took six weeks; code generation took four. More time on the spec than on the code: proof that the spec was the asset, not the output.

The pattern repeats. The expensive, risky part was not generating the new code, but recovering and validating the rules the old system never explained. Where the method helped was in making that work explicit, traceable, and reviewable; where it demanded rigor was in letting no phase pass without a person's sign-off.

07

The risks of decoding with AI are known and *governable*.

Decoding with AI introduces risks of its own. Denying them would be irresponsible; ignoring them, dangerous. Each has a concrete mitigation within the method.

RISK	HOW IT SHOWS UP	MITIGATION
Hallucination about the code	*The AI infers a rule the system never applied*	Traceability back to the source fragment and mandatory human validation before fixing the rule in the spec.
Lost implicit rules	*A rule lives only in an expert's head*	Validation sessions with the key experts during F0–F1, while the knowledge is still available.
Architecture drift	*The generated code drifts from the target design*	Validation of the spec against the target architecture at every phase gate, not at the end.
Functional regressions	*The new system changes a behavior unintentionally*	Test coverage of the legacy behavior, generated from the recovered rules and run on every release.
Transition dependence	*The team cannot run the method without help*	F4 industrializes the practice and transfers the knowledge; the goal is for the organization to sustain it on its own.

08

Four metrics tell you whether the migration is *on track*.

A brownfield migration is not measured by lines migrated. It is measured by knowledge recovered and risk removed. Four indicators are enough to know whether it is heading the right way.

KPI 1	*Rules recovered*	The share of business rules extracted from the legacy and validated by a person. It is the master indicator: it measures the asset, not the code.
KPI 2	*Test coverage*	The share of legacy behavior covered by tests generated from the recovered rules. Without coverage, a recovered rule is an unproven hypothesis.
KPI 3	*Residual debt*	What portion of the system is still undecoded and without an owner. It measures what is left, not what is done; it is the honesty of the project.
KPI 4	*Time-to-first-release*	Time to the first governed release of the new system. It measures whether the method delivers value soon or only promises to deliver it late.

Conclusion

Migrating legacy stops being an act of *faith*.

For too long, modernizing a critical system meant choosing between two bad options: living with a black box or rewriting it blind. AI-assisted engineering opens a third. Recovering the intent the code hides, fixing it in a governable spec, and regenerating against a target architecture turns the migration into a process with traceability, quality gates, and human judgment where the outcome is decided.

The consequence for an organization with legacy is direct. The knowledge that today lives in three people and in uncommented conditionals can become explicit property, readable and executable. The old system stops being a hostage and becomes a source. And the team stops typing to go back to engineering: deciding which rules matter and which test proves it.

The question for any CTO is no longer how much code has to be rewritten. It is another, more uncomfortable and more useful: are your business rules written down somewhere, or do you only believe they are?

Legacy code isn't rewritten. It's decoded into specs and regenerated. The spec is the *asset*; the code, the *by-product*.

No hype. No theory. Just impact.

THINKIA · JUNE 2026

About Thinkia

An AI-native company, not a consultancy with an AI *department*.

Thinkia is a global consultancy born for the Age of Intelligence. Its ontology and DNA are AI-native: strategy, design, and engineering work as a single engine to take the client from idea to a result that operates. We do not hand over technology; we deliver measurable results, and we work across the whole cycle rather than running a pilot and leaving.

In software development, that conviction takes the form of the Coding 3.0 paradigm and the AI-SDLC methodology: specify intent with rigor, validate against architecture, and let the code be a consequence of the spec. This document is one piece of that idea applied to the most persistent problem of any organization with history: legacy.

HOW TO CITE THIS DOCUMENT

Alejano, D. (2026). *Legacy isn't rewritten: brownfield migration with architectural intent*. Thinkia Whitepaper. thinkia.com

REFERENCES

Sources.

Figures and references cited throughout the document. Data from real programs is presented anonymized under confidentiality policy and GDPR.

01 Thinkia internal analysis of AI-SDLC / Coding 3.0 development programs in production, 2025. Development-effort reduction of 37–42 % versus the traditional methodology, while holding quality.

02 Studies of AI adoption in the workplace, 2025. Roughly 75 % of knowledge workers report using AI at work, often without IT oversight (*Shadow AI*).

03 Illustrative model of business-rule recovery by migration approach (Exhibit 01). Thinkia analysis of real programs; the figures represent comparative orders of magnitude, not a single measurement.

thinkia

thinkia.com

© 2026