DEDICATTED

DEDICATTED ENGINEERING GUIDE

Modernization without the three-year program

How agentic AI changes the sequence, the risk and the business case for legacy modernization on AWS

Written for CTOs and heads of engineering carrying an estate they inherited: applications nobody fully documented, licenses nobody wants to renew again, and a modernization business case that has already been turned down twice. The tooling changed in 2025. The sequence you run should change with it.

Dedicatted
AWS Premier Tier Services Partner
Montreal and Toronto

Serhii Semenchenko
Chief Technology Officer, Dedicatted
August 2026 · dedicatted.com

What is in this guide

Thirteen sections. The first three reset the decision model, the middle six cover the work itself, and the last four are what you can start on Monday.

01	**The estate is the problem, not the application**	Executive summary	3
02	**What actually changed in 2025**	Agentic modernization, with numbers	4
03	**Five decisions, not six Rs**	Choosing a path per application	5
04	**Triage the estate first**	Three questions, scored not debated	6
05	**Discovery is what AI actually transforms**	Reverse engineering and business rules	7
06	**Testing is the real constraint**	The arithmetic most plans skip	8
07	**Where the money actually is**	Run cost, licensing and the CFO	9
08	**A worked example**	The shape of the arithmetic	10
09	**Governance for a regulated estate**	Traceability, residency, oversight	11
10	**What agentic modernization will not fix**	The honest list	12
11	**What to measure**	Signals that survive a board review	13
12	**Your first 12 weeks**	A sequenced first move	14
13	**Readiness self-check**	Where you are, and what is next	15

A note on the numbers in this guide

Every figure attributed to AWS comes from AWS's own published announcements and product pages. The worked example in section 08 is an illustrative model built on stated assumptions, not a measured result from any engagement. Where a number is ours, it says so. We would rather hand you arithmetic you can check than a percentage you have to take on faith.

The estate is the problem, not the application

Most modernization business cases get written about one system. They get rejected because the number is large, the payback is distant, and the risk sits entirely on the sponsor. Meanwhile the actual cost is spread across forty applications nobody has looked at properly in years. The estate is where the money is, and it is also where agentic tooling has changed the economics.

1 **Discovery stopped being the expensive part.** Reading an undocumented codebase, extracting what the business rules actually are, and producing current documentation used to take a specialist months. Agents do the first pass in days. That single change reorders the whole plan, because discovery was what made every estimate a guess.

2 **Testing is now the constraint, and the arithmetic is unforgiving.** AWS puts test planning and validation at up to half of a typical modernization timeline. If you speed up everything except testing by five times, the project finishes about 1.7 times faster. That is the entire reason modernization programs disappoint after a promising pilot.

3 **The business case should be built on run cost, not developer productivity.** Productivity gains are real and hard to bank. Licensing, infrastructure and support contracts are lines your CFO already tracks. Build the case there and the approval conversation changes completely.

4 **The cheapest modernization is deletion.** Activity analysis across an estate reliably finds applications that almost nobody uses and that everybody assumed were load-bearing. Retiring those is pure margin, and it buys you political room for the harder work.

Stop asking which system to modernize. Ask which twelve weeks will prove the method.

One application, taken end to end, with the run cost measured before and after. That artifact unlocks the next approval. A three-year roadmap does not.

Who this is for

Technology leaders with a real legacy estate and a real constraint: a data centre lease, a VMware renewal, an audit finding, or a roadmap the current architecture cannot carry. Not for greenfield teams.

What you will walk away with

A triage model for the estate, a clear read on where agents help and where they do not, a business case built on numbers finance recognizes, and a twelve week first move.

What actually changed in 2025

Legacy code is the worst case for a general purpose coding assistant. Those tools learned from clean, documented, conventional repositories. Your estate is the opposite: undocumented dependencies, workarounds nobody wrote down, and behaviour that is correct only because a downstream system expects it. That mismatch is why teams bought licenses and got faster typing rather than faster modernization.

What changed is that purpose-built modernization agents arrived. AWS Transform launched in May 2025 as an agentic service aimed specifically at mainframe, VMware, Windows and custom code, and it has been extended steadily since. These are not chat assistants pointed at old code. They are agents that do code analysis, dependency mapping, business rule extraction, domain decomposition and test generation as distinct, inspectable steps.

1.1bn

Lines of code analyzed with AWS Transform

AWS re:Invent 2025 announcement

810k+

Hours of manual effort saved

AWS's own published figure for the same period

up to 5x

Faster modernization of Windows, mainframe and VMware workloads

AWS Transform product claim

30%

Of a typical team's time spent on manual tech debt work

AWS, on the cost of the status quo

Read the claim precisely

"Up to 5x faster" describes the tasks the agents perform, not your program end to end. The analysis, documentation, planning and transformation steps compress hard. Approvals, data migration windows, user acceptance and regression sign-off do not. Section 06 covers what that does to the overall number, and it is the most important page in this guide.

Why this matters more for legacy than for new build

On a greenfield project, AI mostly saves typing. On a legacy estate it removes the specific bottleneck that made modernization unpredictable: nobody knowing what the system actually does. Once you can read the estate cheaply, you can plan against it. That is a bigger shift than any change in code generation speed.

The pattern is now assessment-to-code, connected

The useful version keeps a chain: activity data informs the retire list, code analysis informs decomposition, extracted rules inform the target design, and generated tests trace back to those rules. Tools that only do one step in isolation leave you doing the expensive integration work by hand.

Five decisions, not six Rs

The migration taxonomies are fine as vocabulary and useless as a decision tool, because they invite an architecture debate per application. Ask what the application is costing you and what it is blocking, and the path usually picks itself.

Retire DO THIS FIRST	Activity data says almost nobody uses it, and the people who do have an alternative. More common than anyone expects.	Agents now produce usage and activity analysis across the estate, so the retire list is evidence rather than a political argument.
Rehost DEADLINE DRIVEN	A date is forcing you: a VMware renewal, a data centre lease, end of support. The code is not the problem this quarter.	Infrastructure migration is heavily automated now. Take the speed, bank the exit, and be honest that you have not modernized anything yet.
Replatform COST DRIVEN	The application works. What hurts is what it costs to run: licenses, oversized hosts, a support contract with one vendor.	This is where the business case is easiest to defend, because the saving lands on a line finance already reports.
Refactor ROADMAP DRIVEN	The architecture is actively blocking things the business has asked for. Every change is expensive and touches too much.	Where agents help most: dependency mapping, decomposition into services, and generating the test coverage that makes the change safe.
Reimagine PROCESS DRIVEN	The business process encoded in the system is itself out of date. Porting it faithfully would preserve the wrong thing.	The newest option. Extract the business rules from the legacy code, put them in front of the people who own the process, then design forward from what they confirm.

Reimagine is the one worth understanding

For thirty years, "what does this system actually do" was answerable only by the two people who had worked on it longest. Rule extraction changes that. You can now put a readable list of encoded business rules in front of the business owner and ask which of them still make sense. Quite often the answer reveals that a third of the system exists to serve a process that ended years ago.

Mixing paths across one estate is correct

A single strategy applied to sixty applications is a sign nobody did the triage. Expect a real estate to split across all five, weighted heavily toward retire and replatform, with refactor reserved for the handful of systems that genuinely block the roadmap.

If everything scores as refactor, the scoring is wrong.

Triage the estate first

Three questions per application. Score them, do not discuss them. The discussion happens once, over the finished scores, and it takes an afternoon rather than a quarter.

What does it cost to run?

Infrastructure, licenses, support contracts, and the engineering time spent keeping it alive. That last one is usually the largest and the least tracked. Ask the team how many days a month go to this system and write the answer down.

What does it block?

Name the specific thing the business asked for that this system made hard, slow or impossible. If nobody can name one, the application is not blocking anything and it does not belong near the top of the list, however old it is.

What does it risk?

Unsupported runtime, a single person who understands it, an open audit finding, data handling that would not survive review. Risk is the argument that moves a board when cost alone does not.

The fourth question, asked only once the first three are scored

Is anyone actually using it? Pull the activity data before you assume. Across a typical estate a meaningful slice of applications turn out to have a handful of monthly users and a documented alternative. Those are not modernization candidates. They are retirement candidates, and retiring them is the fastest return available to you.

Then pick the proof, not the prize

The instinct is to start with the most important system. Resist it. The first application you take end to end should be chosen because it will finish, produce a clean before and after, and be understandable to somebody outside engineering. Its job is to earn the next approval.

Good candidates are mid-sized, have a clear owner, have a measurable run cost, and are not on the critical path for a quarterly close.

What triage produces

- A scored inventory, with one owner named per application.
- A retire list backed by activity data, not opinion.
- A run cost figure per application that finance recognizes.
- One chosen application for the first twelve weeks.

Discovery is what AI actually transforms

If you take one technical point from this guide, take this one. The step that agents change most is not writing the new code. It is reading the old code, and that step was the reason modernization estimates were never trustworthy.

What the agents produce

- **Code analysis.** Languages, volume, complexity, dependencies, and the pieces that are referenced but missing. This runs first and everything else builds on it.
- **Data source mapping.** What stores exist, their metadata, and which parts of the application actually touch them.
- **Business rule extraction.** The encoded logic, written out in language a business owner can read and respond to.
- **Activity analysis.** Static analysis combined with runtime data, so you can see which batch jobs and transactions are genuinely in use.
- **Technical documentation.** Current, generated from the code as it is, not as someone described it in 2014.
- **Domain decomposition.** A proposed split into capabilities and flows, as a starting point for the target design.

Use it as a hypothesis, not a specification

Extracted rules describe what the code does. They do not tell you what it was meant to do, which of it is a bug that became load-bearing, or what the business wants now. Every extracted rule set needs a named human owner who reads it and confirms, corrects or retires each rule. That review is the real work of the discovery phase, and it is the step teams skip when the output looks polished.

Two failure modes

- Treating generated documentation as verified. It is a very good first draft of the truth, produced quickly, and it will contain confident errors.
- Running discovery on the whole estate at once. You get a large volume of output nobody has time to confirm, and confidence in it decays before it gets used.

The bottleneck moved from "nobody knows what it does" to "nobody has confirmed what it should do." That is a much better problem.

Testing is the real constraint

AWS states that test planning and validation traditionally take up to half of a modernization project's timeline. Hold that number next to the acceleration claims and the whole picture changes.

THE ARITHMETIC, ON AWS'S OWN FIGURES

Share of timeline spent on testing and validation	**50%**	Everything else, accelerated by agents	**5x**
Remaining time, as a fraction of today	**0.50 + 0.10**	Actual end-to-end speed-up	**about 1.7x**

Now move testing too. Bring validation down by half and the same project finishes roughly three times faster. The acceleration you can advertise is set almost entirely by what you do with the slowest phase, not by how fast the agents write code.

What to do about it

- Generate test plans, test data collection scripts and automation scripts from the extracted rules, in the same pass. Agents do this now, and it is the highest-value place to point them.
- Trace every generated test back to a confirmed business rule. Untraceable tests are noise that slows sign-off rather than speeding it.
- Build the comparison harness early. On most migrations the honest acceptance criterion is that old and new produce the same output for the same input, at volume.
- Get the acceptance owner to agree what "done" means in week two, not week ten.

The trap worth naming

If your legacy system has no reliable definition of correct behaviour, generated tests will faithfully encode its current bugs and then defend them. You will pass every test and ship the same defects into a more expensive architecture.

Where behaviour is genuinely unknown, the confirmed rule set from discovery becomes the specification, and it needs a business owner's signature before the tests are written against it. There is no tooling shortcut around that signature.

Whatever you do not accelerate sets your ceiling. On modernization programs, that is nearly always validation.

Where the money actually is

Modernization business cases built on developer productivity get rejected, and they should be. The saving is real but it is diffuse, it lands as capacity rather than cash, and finance has heard the pitch before. Build the case on run cost instead. Run cost is a line the CFO already reports on, which means you are asking to reduce a number they can see rather than to believe in one you have modelled.

Operating system and runtime licensing

The single clearest line. AWS states that moving from .NET Framework to cross-platform .NET removes Windows license dependencies and can cut operating costs by up to 40% on those workloads. That is a recurring saving, it starts the month you cut over, and it needs no assumption about how fast anyone codes.

Virtualization and the VMware question

Licensing changes over the last two years turned a quiet renewal into a board-level decision at a lot of organizations. If you are facing one, that renewal date is the strongest forcing function you will get. Use it, and be clear internally that a rehost buys you the exit rather than the modernization.

Commercial middleware and support contracts

Application servers, message brokers, schedulers and the support agreements around them. These rarely appear in the engineering conversation and often carry more annual cost than the infrastructure underneath them. Pull the contract list before you scope anything.

Capacity you are paying for and not using

Legacy estates are sized for a peak that was estimated once and never revisited. Containerization and managed services make right-sizing real rather than theoretical, and this is usually the fastest saving to demonstrate on the first application.

And the one that is not a saving, but wins the argument anyway

Retirement. Every application you switch off removes its infrastructure, its licenses, its support contract and the engineering days it quietly consumed every month. It is the only item on this page with no delivery risk attached, and it is the reason activity analysis belongs at the start of the program rather than somewhere in phase two.

Say the payback period out loud

A recurring saving against a one-off investment gives you a payback in months, and that is the number an approver actually wants. If your payback is longer than twenty-four months on the first tranche, you have probably picked the wrong applications to start with rather than the wrong strategy.

Then treat productivity as the upside

Faster delivery, current documentation and a team that can change the system without fear are genuinely valuable. Present them as what you also get, after the run cost case has already cleared the bar. That ordering has a much better approval rate than the reverse.

The shape of the arithmetic

An illustrative estate, so you can see how the pieces combine. The assumptions below are stated rather than measured. Replace every one of them with your own before this goes near a finance review.

ASSUMED ESTATE

Applications in scope	**60**	On .NET Framework or Windows Server	**18**
On virtualized on-premises infrastructure	**12**	Retirement candidates found by activity analysis	**9**
Genuinely blocking the roadmap	**4**	Leave alone this year	**17**

HOW THE FIRST TRANCHE RESOLVES

Retire the 9. Saving	**100% of their run cost**	Replatform the 18 off Windows licensing	**up to 40% (AWS)**
Rehost the 12 ahead of the renewal date	**exit, not saving**	Refactor the 4, sequenced one at a time	**roadmap unblocked**

Twenty-seven of sixty applications produce a recurring saving without anyone refactoring anything. That is the tranche that funds the interesting work, and it is the one most roadmaps put last because it is less interesting to engineers.

Why the order matters this much

Run the retire and replatform tranche first and you reach the refactor conversation with a demonstrated saving and a team that has done it once. Run refactor first and you reach month nine with a large invoice and nothing to show a board.

What this model does not claim

No productivity multiplier, because those are the numbers that fall apart under scrutiny. Everything above is infrastructure, licensing and contracts. If the case clears on those alone, productivity becomes upside rather than a load-bearing assumption.

CLIENT OUTCOME · DEDICATTED

One platform, moved for the licensing

A global security services company ran its workforce scheduling platform on OpenShift. The application worked. What hurt was what it cost to run: a Red Hat subscription that renewed every year regardless of what the platform delivered. We moved it to Amazon EKS. The scheduling logic was not rewritten and the business case never rested on anyone shipping faster. It rested on a subscription that stops.

Amazon EKS	**~$1M**	**June 2026**
Migrated from OpenShift	Red Hat licensing retired	Live in production

Governance for a regulated estate

If your estate carries regulated data, the questions below get asked before anyone approves a pilot. Having answers ready is the difference between a four week procurement conversation and a four month one.

Where does your code go?

Agents read your source. Know which service processes it, in which region, under what retention terms, and whether any of it can be used for training. For a regulated workload this is a procurement question before it is a technical one, and the answer needs to be in writing.

Traceability, end to end

A failing test should trace to a rule. The rule should trace to a line in the legacy system and to the owner who confirmed it. Keep that chain and an audit of the modernized system becomes a document retrieval exercise. Break it and you are reconstructing intent under time pressure.

Human approval at named gates

Agents propose, people approve. Define the gates explicitly: the confirmed rule set, the target design, the cutover plan. Log who approved what and when. This is also what makes the work defensible internally when something eventually goes wrong.

Standards carried into the agent

Put your security policy, approved dependency list and architecture standards into the agent's operating rules rather than checking for them at review. Generated code that never violates the standard is cheaper than generated code you have to inspect for violations.

Data migration is where regulated programs actually get stuck

Code transformation is now the tractable part. Moving regulated data, proving it arrived intact, keeping both systems consistent through a parallel run, and satisfying a regulator that nothing was exposed in transit is slower than any of it and does not compress with agents. Scope it as its own workstream with its own owner, and start it earlier than feels necessary.

Residency shapes the architecture, not just the paperwork

Which regions can hold the data, which can process it, and whether that differs for backups and logs. These constraints are much cheaper to satisfy in the target design than to retrofit after a cutover, and they routinely rule out the simplest option.

Keep the old system warm longer than planned

A parallel run is expensive and it is the cheapest insurance available on a regulated cutover. Budget for it explicitly rather than treating it as contingency, and decide in advance what evidence ends it.

What agentic modernization will not fix

The tooling is genuinely better than it was eighteen months ago. It is not better at any of the following, and every stalled program I have seen stalled on one of them rather than on the technology.

1. **Business intent nobody recorded.** An agent will tell you precisely what the code does. It cannot tell you what it was supposed to do, which behaviour is a defect that downstream systems now depend on, or what the business would want if you asked today. That still requires a person with authority and institutional memory, and if that person is not available, no amount of tooling substitutes.

2. **An estate nobody owns.** If you cannot name one accountable person per application, you do not have a modernization problem yet. You have an ownership problem, and starting the technical work first means every decision escalates and nothing closes.

3. **Testing discipline you never had.** Generated tests written against a system with no agreed definition of correct behaviour will encode the current bugs and then protect them. You will pass everything and ship the same defects into a newer, more expensive architecture.

4. **Organizational appetite.** Modernization asks people to accept short-term disruption for a benefit that arrives later and mostly accrues to somebody else's budget. That is a sponsorship problem. It is solved with evidence and a named executive owner, not with a better roadmap document.

Every one of these predates AI, and every one of them is now the binding constraint.

That is not a reason for pessimism. It means the technical risk has genuinely come down and the remaining work is organizational, which is work you already know how to do.

A pattern worth watching for

Discovery output arrives fast and looks authoritative, so teams skip the confirmation step and build against it. Six weeks later somebody notices a rule was misread and the target design has to move. Budget review time for every rule set, and treat a confirmed rule as a deliverable in its own right.

And one worth insisting on

Nothing gets modernized until somebody's name is against it. One owner per application, written down, agreed by that person. It is the least technical item in this guide and the strongest predictor of whether a program finishes.

Signals that survive a board review

Pick metrics an approver can check independently. Anything that depends on an engineer's estimate of their own speed will be discounted, correctly, the moment it is challenged.

Applications retired

The cleanest number you have. It is verifiable, it has no delivery risk attached, and it maps directly to contracts that stop.

Run cost per application, monthly

Infrastructure, licensing and support, tracked per application before and after. This is the line the business case was built on, so keep reporting it.

Lead time for a change

Modernized workloads against legacy ones, measured the same way. This is what "unblocking the roadmap" looks like in numbers.

Change failure rate

The counterweight. Faster delivery that raises failure rate has moved cost rather than removed it, and it will be noticed.

Rules confirmed by an owner

Of the business rules extracted, how many has a named human actually reviewed and signed off. This is your real progress indicator during discovery.

Estate with current documentation

Share of in-scope applications with generated, current technical documentation. A good proxy for how much key-person risk you have removed.

Two rules that matter more than the list

Capture the before. A run cost figure taken after you started is not a baseline and will not survive a question.
Report per application, not in aggregate. Portfolio averages hide both the wins that justify the next tranche and the applications quietly going wrong.

What not to lead with

Lines of code transformed, hours saved, story points, adoption percentages. These are useful internally and they are weak in front of a board, because none of them is a number the organization was already managing. Keep them in the appendix.

Then put it in the next case

The first tranche exists to produce evidence. When you go back for the second, you are no longer presenting a model. You are presenting a measured result from your own estate, and that is a materially easier conversation.

Your first 12 weeks

One application, taken all the way through, with the numbers captured at both ends. The output is not a modernized system. It is the evidence that buys the next approval.

1

WEEK 1 TO 2

Inventory and activity data

List everything, including the applications that are somebody's side responsibility. Pull real usage data rather than asking people what they think is used. Name one owner per application and get that person to agree in writing.

2

WEEK 2 TO 4

Score, triage and choose

Cost, blocking, risk. Score first, discuss once. Publish the retire list and start switching those off immediately, because it is the fastest return available and it demonstrates momentum while the rest is still being planned.

3

WEEK 4 TO 6

Discovery on one application

Run code analysis, rule extraction and documentation generation on the chosen system. Then do the part that matters: put the extracted rules in front of the business owner and get each one confirmed, corrected or retired. Treat the confirmed set as the specification.

4

WEEK 6 TO 10

Build, test and cut over

Generate tests from the confirmed rules and keep the trace between them. Stand up the comparison harness against the legacy system. Migrate, run in parallel, and hold the parallel run until the evidence you agreed in advance says you can stop.

5

WEEK 10 TO 12

Package the evidence

Run cost before and after. What worked, what cost more than expected, what you would sequence differently. Write it for a finance audience, not an engineering one. This document is the actual deliverable of the first twelve weeks.

Resist adding a second application before week twelve.

The value of this quarter is a clean, defensible before and after. Running two in parallel gets you two partial results and no evidence.

Readiness self-check

Tick honestly. The group with the most gaps tells you where the first twelve weeks should actually go.

Know your estate

- Every in-scope application has one named, agreed owner.
- Run cost is known per application, including licenses and support contracts.
- Real usage data exists, so the retire list is evidence rather than opinion.
- You can name what each priority application is blocking.

Ready to defend it

- The business case rests on run cost, with productivity as upside rather than as load-bearing.
- A before-state baseline is captured and dated.
- You know where your source code is processed and under what retention terms.
- Traceability runs from confirmed rule through code to test, and is kept.

Ready to execute

- One application is chosen for the first tranche, picked to finish rather than to impress.
- A business owner is available to confirm extracted rules, with time booked.
- Acceptance criteria are agreed before the build starts.
- A comparison harness against the legacy system is planned, not assumed.

How to read your score

Gaps mostly in "know your estate". Spend the first twelve weeks on inventory, ownership and activity data. Do not start a migration yet.

Gaps mostly in "ready to execute". You have the map. What is missing is a business owner with booked time and an agreed definition of done.

Gaps mostly in "ready to defend it". The engineering will go fine and the second tranche will not get funded. Fix the baseline first.

The hard part moved

For most of the last two decades, the reason legacy modernization was risky is that nobody could read the estate cheaply. Every estimate was a guess wrapped around an unknown, and the honest response was a long program with a large contingency. That is the thing that actually changed.

What is left is validation, ownership and organizational appetite. None of those compress with better tooling, and all three are problems you already know how to run. The teams that get value here are the ones that stop treating modernization as a technology program and start treating it as a portfolio decision with a technology component.

Pick one application. Take it all the way through. Measure both ends. Then go back and ask for the rest.

Read the source

AWS Transform product page and the re:Invent 2025 announcement, for the agent capabilities and the published figures used throughout this guide.

The delivery side

Our companion guide covers AWS AI-DLC: how to run an AI-driven lifecycle once the estate work is under way. Ask us for a copy.

Talk to us

We modernize AWS estates as an AWS Premier Tier Services Partner, including regulated environments where the audit trail has to hold up.

Bring us your estate list. We will score it with you.

In 60 minutes we will run the three questions across your applications, mark the obvious retirement candidates, and name the one system worth taking end to end this quarter. You keep the scored list either way.

Send the list, or just the application count, to:

contact@dedicatted.com →

dedicatted.com

Figures attributed to AWS are drawn from AWS's published product pages and announcements, including the AWS Transform re:Invent 2025 announcement. The worked example in section 08 is an illustrative model built on stated assumptions and is not a measured outcome from any engagement. Dedicatted is an AWS Premier Tier Services Partner. This guide is Dedicatted's own point of view and is not an AWS publication.