forgeahead

Enterprise Cloud Modernization Guide



From Legacy Systems to AWS

A practical guide for technology leaders planning AWS cloud modernization, and getting the data ready for AI

Table of Contents

Executive Summary 03

Chapter 1: The Hidden Cost of Legacy Systems in the Enterprise 04

Chapter 2: What Modernization Actually Means on AWS 07

Chapter 3: The Migration Roadmap: A Leader's Playbook 10

Chapter 4: From Modernized Applications to an AI–Ready Enterprise 13

Chapter 5: Architecture Patterns That Work 15

Chapter 6: Measuring What Matters: ROI and KPIs 17

Chapter 7: Choosing the Right Modernization Partner 19

Get Started 19

References 20

Executive Summary

Most big companies have one system everyone is a little afraid of. Billing, claims, order management, whatever it is in your shop. It's old, it's been patched for years, and a couple of people really understand it. If one of them leaves, you have a problem.

This guide is for whoever has to decide what to do about it. You've probably agreed it needs to move already. What's still open is the order: what goes to AWS as it is, what gets rebuilt, what gets switched off, and how you do all that without stopping the business.

We go through it in that order. First, what the old systems really cost. Then how to build a cloud modernization strategy that survives contact with your actual portfolio, how to run the move in waves, and what the work does for your data when you want to start on AI. It comes from the cloud migration and modernization projects our engineers have delivered on AWS. Where we got things wrong, we say so.

If you only read one chapter, read Chapter 3. That's where most programs go sideways.



CHAPTER 1

The Hidden Cost of Legacy Systems in the Enterprise

Built for a business that no longer exists

Most legacy systems were built for a simpler setup. People worked from offices, releases happened twice a year, and systems often exchanged data through a flat file sitting on an FTP server overnight. One customer. One main channel. That model worked then. It gets harder to defend as the business changes.

Now the same application has to feed a mobile app, a partner API, a reporting team, and an AI assistant somebody saw at a conference. It was never designed for any of that. The database is shared with five other systems and the integrations are point to point, so every new request becomes its own project, with its own estimate and its own fight over whose budget pays. That's usually when legacy application modernization finally gets a line in the plan.

Where the money actually goes



Nobody can give you one number, because the cost is spread over five or six budgets.

Run cost is the obvious one: hardware refresh, licences, data center space, all sized for your busiest day. Change cost is next. A two-day fix takes three weeks because nobody is sure what else it touches. Then there are people. When three engineers carry all the knowledge, they know it, and so does HR.

Audit costs add up too. The same end-of-support operating system and the same unpatched middleware show up on the report every quarter, and every quarter someone signs a risk acceptance.
The last one is the cost nobody counts: the analytics and automation ideas that never happened because the data was stuck inside a system that couldn't hand it over.

Add it up and the old estate is competing with new projects for the same money. It usually wins.

Why waiting costs more

Waiting doesn't stop the bill. Every workaround you add this year is something to untangle next year.

The bigger problem is that end-of-support dates and contract renewals eventually set your timeline for you. Decisions made with ninety days left on a data center lease tend to be bad ones. You rehost everything in a hurry, hit the date, and then run data center designs at AWS prices for years. [Replace this with a real Forgeahead example if you have one.]

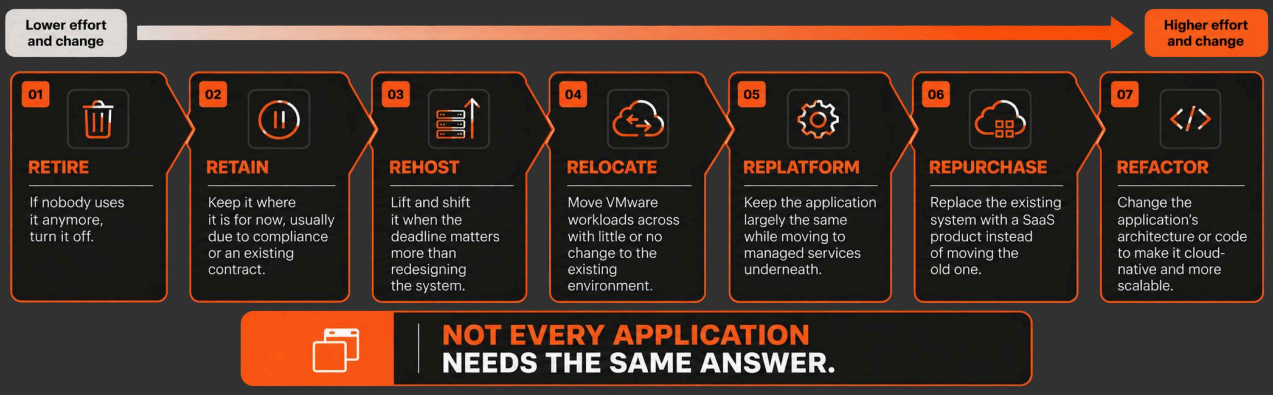
An AWS modernization program that starts while there's still time to think almost always costs less than the emergency version.

CHAPTER 2

What Modernization Actually Means on AWS

One portfolio, several different answers

Don't treat the estate as one project. It's usually a few hundred small ones. A handful deserve a rebuild, most just need a better home, and more than you'd expect can be switched off.



The options are usually called the 7 Rs:

- Retire: if nobody uses it anymore, turn it off.
- Retain: keep it where it is for now, usually because of a compliance requirement or an existing contract.
- Rehost: lift and shift it when the deadline matters more than redesigning the system.
- Relocate: move VMware workloads across with little or no change to the existing environment.

- Replatform: keep the application largely the same while moving to managed services underneath it. Moving a self-managed Oracle or SQL Server database to Aurora or RDS is a common example.
- Repurchase: replace the existing system with a SaaS product instead of moving the old one.
- Refactor: change the application's underlying architecture or code to make it more scalable, maintainable, or cloud-native, without changing what the application fundamentally does.

Real portfolios are usually a mix. A few core systems may need to be refactored. More will make sense to replatform and the retire pile is often bigger than expected. Once you map that out, you have the beginnings of a cloud modernization strategy. If the plan says "refactor everything," the numbers probably haven't been worked out. Neither has the delivery plan.

Moving a workload is not the same as modernizing it

Move a monolith to EC2 and you have moved the workload. The application still deploys the same way, has the same failure points, and struggles with the same traffic spikes.
Modernization means changing how the application is built and operated. That could mean splitting a large codebase around business capabilities, so the billing team can release changes without waiting for the catalog team. It could mean replacing infrastructure that the team has been maintaining themselves with managed services. It could also mean automating deployments and recovery so the process does not depend on someone remembering a series of steps at 11 pm.

On AWS, that might mean ECS or EKS for long-running services, Lambda for event-driven workloads, Aurora or DynamoDB in place of databases that need regular manual maintenance, and S3 for storage without having to plan capacity upfront.

And keep the platform choice practical. ECS is enough for many teams. EKS makes more sense when there is already a team that knows Kubernetes and actually needs what it provides.

Elastic capacity changes the math

Traditional infrastructure is bought for the worst day of the year, and you pay for it on the quiet days too. Elastic capacity follows demand instead. The catch is that the application has to scale down as well as up, and plenty can't. The savings only show up once an engineer has done that work.

Rehosting still has a place

I'm not against lift and shift. With a hard data center exit date, it's often the right call. Just put the second pass on the plan with an owner and a date. Without those, "temporary" becomes permanent.

CHAPTER 3

The Migration Roadmap: A Leader's Playbook

How to modernize legacy applications on AWS

People ask us how to modernize legacy applications on AWS as if there's a trick. There isn't. It's four phases, in order, and each one lowers the risk for the next.

Phase 1 : Discover and assess

Start with an inventory of what you already have. List the applications, who owns them, what they connect to, who uses them, and what they cost to run. AWS Application Discovery Service and Migration Hub can help map servers and network connections, but that only gives you part of the picture.

The rest usually comes from the people running these systems. Ask the ops team what tends to break at month end. You may hear about a spreadsheet that gets sent to finance every Friday, or a croon job running on a server nobody remembers setting up. Those dependencies rarely show up in a discovery tool.

Then make a call on each application: retire, retain, rehost, replatform, or refactor. Don't spend a week debating every application. Make the call, record the reason, and move on. You can revisit it later.

That inventory becomes the working list for the AWS modernization effort. Get it in place before you start moving anything.

Phase 2 : Build the foundation

Build the landing zone before any workload moves. That means accounts, networking, identity, logging, guardrails, and cost controls. Control Tower and Organizations are a sensible place to start.

Tag everything from day one. Adding cost allocation tags to a few hundred resources after the fact is miserable. Connect the existing data center over Direct Connect or a VPN, and expect Direct Connect to take weeks to provision, so order it early.

The business will think nothing is happening during this phase. Do it anyway. The alternative is security findings and a cloud bill nobody can explain.

Phase 3 : Migrate in waves

Sort applications by dependencies and risk, then start small. The first wave should be straightforward. Use it to get the team comfortable with the process and find gaps in the runbook before the harder migrations begin.

Application Migration Service handles the server migration, while Database Migration Service handles the data. Start replication early so there is less to move during the actual cutover. If you're changing database engines, such as moving from Oracle to PostgreSQL, give the schema conversion enough time. It usually takes more work than the demo suggests.

For applications that need to be refactored, we usually use the strangler fig pattern. Put a routing layer in front of the existing application, build the new services alongside it, and move functionality across one piece at a time. The old application continues serving production while the new pieces are being introduced.

Every migration wave should have a rollback plan and a clear go or no-go decision. Write the rollback steps before the migration starts, and keep the source environment running until the business has signed off.

Phase 4 : Optimize and switch off

When workloads settle, look at right-sizing, storage tiers, Savings Plans, and monitoring. This is also when you decommission the old environment. Set the shutdown date early and stick to it, or you'll end up paying for both.

The people side

Technology rarely sinks these programs. Unclear ownership does. So does training that never happened, and ops teams who feel the change is being done to them.

Name an owner for every application in every wave. Put training in the budget. Bring ops into design early, because they already know where the problems are.

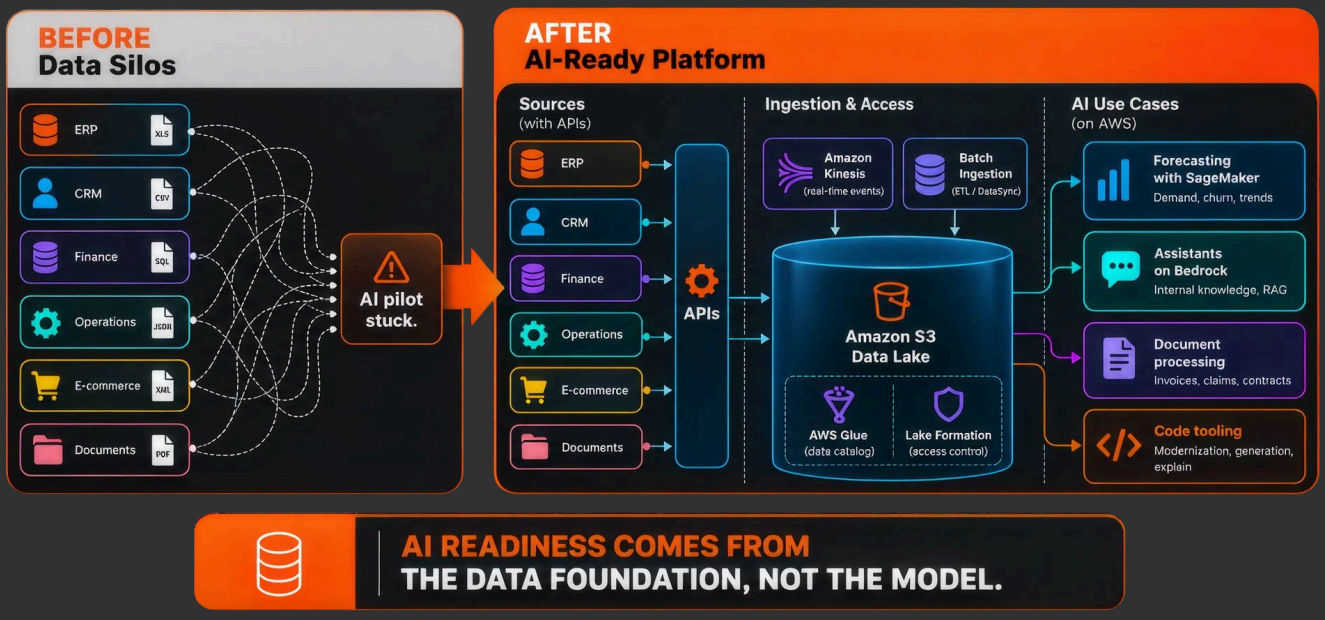
CHAPTER 4

From Modernized Applications to an AI-Ready Enterprise

Why AI pilots stall

Lots of enterprises have run an AI pilot that looked great in a demo and went nowhere. The model is rarely the problem. The data lives in six systems, in four formats, behind integrations someone built in 2014. Getting it somewhere a model can use it takes months.

What changes once the foundation is there



A solid cloud migration and modernization program fixes much of this as a side effect. Applications expose APIs. Operational data lands in an S3 data lake with proper access control, usually through Glue and Lake Formation. Kinesis carries events in near real time. Analytics and ML tools work from current data instead of last month's extract.

The use cases are fairly practical:

- Forecasting and anomaly detection on operational and financial data using Amazon SageMaker.
- Assistants built with Amazon Bedrock that can answer questions using your internal documents.
- Document processing for invoices, claims, contracts, and other work that still involves manual data entry.
- Developer tools that help teams understand and untangle older code before the next stage of modernization.

Governance needs to be part of the plan

Generative AI can produce incorrect answers with a lot of confidence. Without the right access controls, sensitive information can also end up in prompts. Decide the basics early: how data is classified, which outputs need human review, what gets logged, and which models can be used for different types of work.

It is easier to put these controls in place while the systems are being modernized than to add them after an incident.

Modernization should account for AI

Running AI as a separate program can mean doing the same data and integration work twice. If the data architecture is considered as part of the AWS modernization work, the first AI use case has the underlying infrastructure it needs.

That does not mean every modernization project needs an AI use case immediately. It means the architecture should not create unnecessary work when one comes along.

CHAPTER 5

Architecture Patterns That Work

The reference pattern

Most estates we work on end up looking pretty similar once the dust settles.

- Separate AWS accounts for production, non-production, security, and shared services.
- Direct Connect or a VPN back to the data center, with a plan to switch it off eventually.
- VMs for rehosted systems, containers for services, Lambda for the small event-driven jobs.
- Aurora, DynamoDB, or RDS for data, with DMS handling the move.
- API Gateway and EventBridge or SQS between systems, so nobody is reading another team's database.
- CloudWatch and tracing on from day one.

If you're stuck on how to modernize legacy applications on AWS at the design level, this is a fair place to start. Change it to fit your situation.

Security and compliance belong in the design

Build least privilege, encryption, network segmentation, and automated compliance checks into the landing zone at the start. Adding them after a review flags problems takes longer, and everyone hates it. Use the Well-Architected Framework as your checklist. Security and the architects will still disagree, but at least it'll be about the same document.

Resilience, tested for real

Plenty of old DR plans amount to a second data center nobody has failed over to in years. On AWS, multi-AZ and automated backups are easy to get, and a cross-region test doesn't cost much. Run one. Set the recovery targets with the business first, then choose the setup that meets them.

Room to keep changing

A clean design pays off the next time something changes. A new channel or partner integration gets added on top of what's there, and you don't have to reopen the foundations.

CHAPTER 6

Measuring What Matters: ROI and KPIs

Cost and infrastructure

Track cost per transaction, per user, or per business unit. Total cloud spend will climb as the business grows, so it doesn't say much by itself.

Then work out what the old setup really cost: facilities, licences, hardware refresh, the contractor you keep on for that one old system. Do this before the first wave moves. After that, nobody can rebuild the numbers.

Engineering and operations

Look at how often you deploy, how long a change takes to reach production, how many changes fail, and how fast you recover. Also ask your engineers how much of their week goes on keeping things running. If that number drops, that's your real win, even though it rarely shows up in a business case.

Risk and resilience

Count the unsupported systems you've shut down, the critical vulnerabilities still open, and the audit findings closed. Run a real recovery test and write down how long it took. None of this makes the headline, but it's often what gets the budget approved.

Business outcomes

Time to market, customer satisfaction, and new revenue take a while to move. Tell people that at the start. Report the trend over a few quarters, because one before-and-after number proves very little.

Be honest about the investment

This costs real money. Discovery, tooling, a stretch of paying for two environments, training, and sometimes a higher bill for a while. Put all of it in the business case, with dates for when the benefits arrive. Finance will trust it more.

CHAPTER 7

Choosing the Right Modernization Partner

What to look for

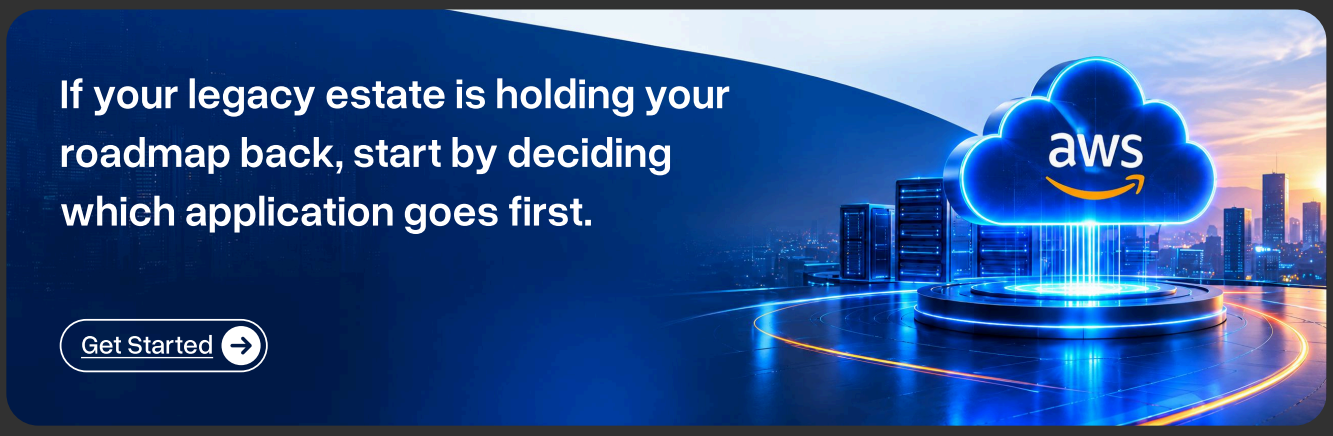
Most programs that struggle fail in delivery. The timeline had no slack. Everything hung on one cutover date. The partner handed over a strategy deck and left before anyone had to run it.

So look for a team that has done the hands-on AWS work, ideally in your industry, on systems as old and messy as yours. They should work in waves and teach your people as they go.

Ask awkward questions. How do you deal with undocumented dependencies? Can I see a rollback plan from an old project? How do you choose between replatforming and refactoring? A specific answer beats a polished slide.

Why Forgeahead

Forgeahead helps enterprise engineering leaders with AWS cloud modernization, from portfolio assessment and landing zone design through wave-based migration and the data work AI needs later. Assess first, migrate in waves, plan the data early. That's how we run it.



Forgeahead plans and delivers cloud modernization strategy, application modernization, and legacy application modernization programs on AWS. Get in touch and we will go through your portfolio and pick a first wave together.

References

- AWS Application Discovery Service. Amazon Web Services. https://aws.amazon.com/application-discovery/
- AWS Migration Hub. Amazon Web Services. https://aws.amazon.com/migration-hub/
- AWS Application Migration Service. Amazon Web Services. https://aws.amazon.com/application-migration-service/
- AWS Database Migration Service. Amazon Web Services. https://aws.amazon.com/dms/
- AWS Control Tower. Amazon Web Services. https://aws.amazon.com/controltower/
- AWS Direct Connect. Amazon Web Services. https://aws.amazon.com/directconnect/
- Amazon Elastic Container Service (ECS). Amazon Web Services. https://aws.amazon.com/ecs/
- Amazon Elastic Kubernetes Service (EKS). Amazon Web Services. https://aws.amazon.com/eks/
- AWS Lambda. Amazon Web Services. https://aws.amazon.com/lambda/
- Amazon Aurora. Amazon Web Services. https://aws.amazon.com/rds/aurora/
- Amazon DynamoDB. Amazon Web Services. https://aws.amazon.com/dynamodb/

References

- Amazon S3. Amazon Web Services. https://aws.amazon.com/s3/
- Amazon Kinesis. Amazon Web Services. https://aws.amazon.com/kinesis/
- AWS Glue and AWS Lake Formation. Amazon Web Services. https://aws.amazon.com/lake-formation/
- Amazon SageMaker. Amazon Web Services. https://aws.amazon.com/sagemaker/
- Amazon Bedrock. Amazon Web Services. https://aws.amazon.com/bedrock/
- Amazon CloudWatch. Amazon Web Services. https://aws.amazon.com/cloudwatch/
- AWS Well-Architected Framework. Amazon Web Services. https://aws.amazon.com/architecture/well-architected/