

MODERNIZING DIGITAL PUBLISHING ON AWS

A Leader's Guide to Legacy CMS Modernization and AI-Powered Content Personalization

Table of Contents

• Executive Summary	03
• Chapter 1: The Hidden Cost of Legacy CMS in Publishing	04
• Chapter 2: What Modernization Actually Means on AWS	06
• Chapter 3: The Migration Roadmap: A Leader's Playbook	08
• Chapter 4: AI-Powered Content Personalization	10
• Chapter 5: Architecture Patterns That Work	12
• Chapter 6: Measuring What Matters: ROI and KPIs	13
• Chapter 7: Choosing the Right Modernization Partner	15
• Get Started	15
• References	16

Executive Summary

Nobody chose the technical debt they're carrying. The CMS that felt cutting-edge a decade ago is now the reason a new feature takes three months instead of three weeks. It's the reason personalization has been "on the roadmap" for two years running. Engineering leaders know this. They live it every sprint.

This guide is for the CTOs, VPs of engineering, and heads of digital who've already had the "should we modernize" conversation and are now stuck on the harder one: how, and in what order. We'll walk through why legacy CMS platforms cost more than they look like they cost, what a modernized architecture on AWS actually involves, a migration approach that doesn't require taking the site down for a weekend, and how AI-driven personalization fits once the underlying data problem is solved.

Everything here comes out of work Forgeahead has done helping publishing engineering teams move off legacy platforms and onto AWS, and layer in the AI capabilities those old systems were never built to support.

Chapter 1

The Hidden Cost of Legacy CMS in Publishing

Where Legacy CMS Really Costs You



Slower Time to Market

New features wait on tightly coupled code before they ship.



Idle Infrastructure Spend

Servers sized for peak traffic sit underused most of the year.



Shrinking Talent Pool

Fewer engineers can maintain aging platforms, and they cost more.



Stalled AI and Personalization

No clean pipeline to feed real-time reader data into recommendations.



Compounding Technical Debt

Small, manageable costs stack into one that eats the engineering budget.

The system was built for a different job

Most legacy CMS platforms in publishing share one assumption baked into their architecture: content goes to a website. That was a fine assumption when the website was the whole product. It stopped holding up once the same article also needed to reach a mobile app, a syndication partner, a smart TV feed, and a newsletter, each with a different shape and different rules.

Monolithic platforms bundle content storage, business logic, and presentation together. Touch one and you risk breaking the other two. A new integration waits for the next release cycle because there's no clean seam to plug it into. None of this was a mistake when the system was built. It's just what happens when a platform outlives the web it was designed for.

Where the cost actually shows up

Ask an engineering leader what a legacy CMS costs and you'll usually get a shrug, not a number. That's because the cost hides in a dozen small places instead of one line item. A feature that should take a week takes six because someone has to trace through code nobody's touched since 2019. Servers stay provisioned for a traffic spike that happens twice a year, so they sit mostly idle the other 363 days. The two engineers who actually understand the platform's quirks become a single point of failure, and they know it, which is part of why they're expensive to keep.

Then there's the cost that doesn't show up on any dashboard: the personalization project that's been "next quarter" for six straight quarters, because nobody's built a way to get real-time reader behavior into a recommendation engine. Add it up and legacy infrastructure isn't just slower. It's actively competing with the engineering budget for resources that should be going toward things that actually move the business.

Why "eventually" is not a strategy

Every team that pushes modernization off another year isn't avoiding the cost. They're financing it. Each new integration bolted onto the old system is one more thing to untangle later. Each year without real architectural investment widens the gap between what a modern reader expects and what the platform can actually deliver, while competitors who already made the move pull ahead on speed and cost.

Chapter 2

What Modernization Actually Means on AWS

What Changes When You Modernize



Legacy CMS

- Content tied to one presentation layer
- Fixed servers sized for peak traffic
- Same monolith, just relocated



Modernized on AWS

- Structured content delivered via API to any channel
- Auto-scaling infrastructure that flexes with demand
- Re-architected around managed services

Decoupling content from presentation

The real starting point for CMS modernization isn't picking a new vendor. It's separating how content is stored from how it's presented. A headless or hybrid-headless setup treats content as structured data, exposed through an API, so the same article can power a website, an app, and a partner feed without three separate builds. Everything else in this guide depends on getting this part right.

On AWS, that usually looks like the CMS layer, whether that's a managed headless platform or something custom, running on Amazon ECS or EKS, with structured content and media sitting in Amazon S3 and a managed database like Aurora or DynamoDB. CloudFront handles global delivery so readers get fast load times whether they're in Mumbai or Minneapolis.

Elastic infrastructure instead of fixed capacity

Publishing traffic doesn't behave like most software. A single breaking story can push traffic to ten times normal for a few hours, then drop right back down. Fixed servers either can't absorb that spike or sit oversized and expensive for the rest of the month waiting for it to happen again. Auto-scaling, Lambda for event-driven work, and CloudFront's edge caching handle the spike automatically and scale back down without anyone getting paged at 2 a.m.

A platform, not just a migration

Here's the mistake worth naming directly: lifting the same monolithic system onto AWS servers and calling it modernized. That keeps every architectural constraint the platform already had, just on someone else's hardware. Real modernization rebuilds around managed services and decoupled delivery, which is more work up front, and it's the difference between a platform that holds up for five years and one you're modernizing again in eighteen months.

Chapter 3

The Migration Roadmap: A Leader's Playbook

Four Phases, Zero Blackouts



Assess

Map content, integrations, and real bottlenecks before touching architecture.



Decouple in Place

Build the new layer alongside the old one, no forced cutover.



Migrate Incrementally

Shift content and traffic in stages, validating as you go.



Optimize

Retire the legacy system, tune performance, unlock cost savings.

Phase 1: Assess before you architect

Every modernization effort should start with a genuinely honest audit: what content types exist, how they're structured, which integrations quietly depend on the current system, and where the real bottlenecks sit versus where people assume they sit. Skip this step and you'll find the dependency nobody mentioned halfway through the migration, usually at the worst possible time.

Phase 2: Decouple in place

Rather than a single risky cutover, the more reliable path runs API-driven content delivery alongside the existing CMS before retiring anything. Engineering can build and test new front-end experiences against real content while the legacy system keeps serving production traffic. Editorial teams get time to adjust to workflow changes without a hard deadline forcing the issue.

Phase 3: Migrate content and traffic incrementally

Migrate by content type or by section, not all at once. Each stage confirms structured content renders correctly, redirects preserve SEO value, and editorial workflows still work before the next section moves. Traffic can shift the same way, with weighted routing sending a small slice of readers to the new platform first and growing that share as confidence builds.

Phase 4: Decommission and optimize

Once the new platform reliably carries full production traffic, the legacy system can finally be shut down. Attention turns to tuning caching, right-sizing compute, and building the pipelines personalization depends on. This is also where the savings from elastic infrastructure start showing up against the old fixed-capacity spend.

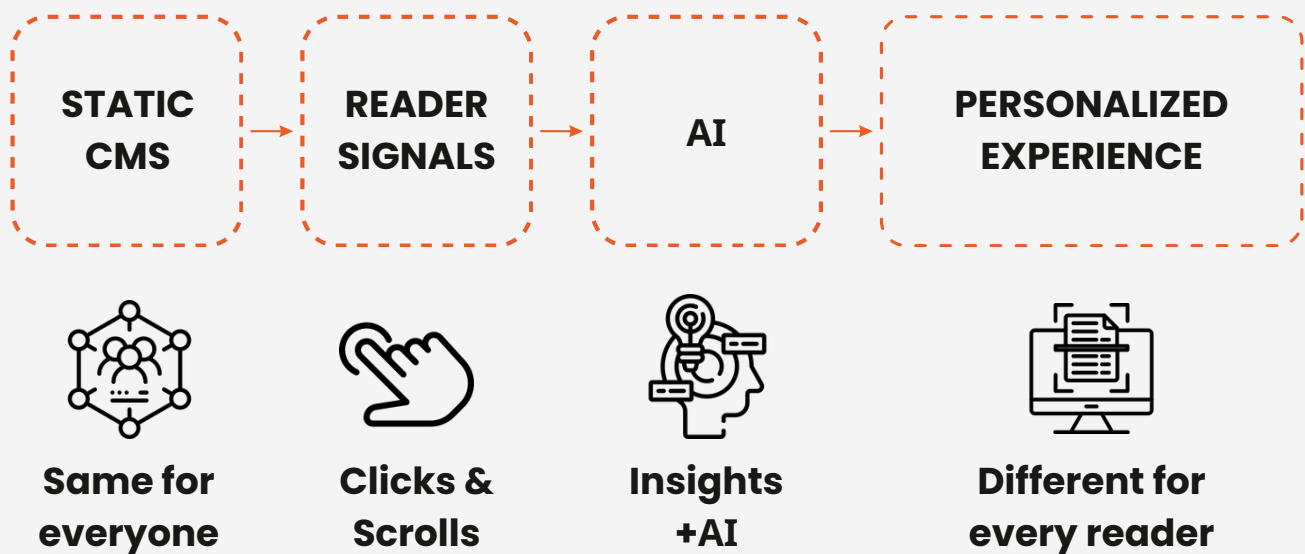
What makes this approach work

Nothing here forces a content blackout or a single high-stakes cutover. Readers don't notice the migration happening. The editorial keeps publishing on schedule. Engineering validates each stage before committing to the next, which is what makes the whole plan realistic for a publisher that genuinely can't afford downtime.

Chapter 4

AI-Powered Content Personalization: From Static Pages to Dynamic Experiences

From Static to Adaptive



Why personalization has been out of reach

Most legacy CMS platforms serve the same homepage, the same recommended articles, and the same email digest to every single reader. Personalization has been on the roadmap at most publishers for years, and it keeps stalling for the same reason: the platform has no real-time pipeline connecting what a reader does to what content decisions get made. Bolt a recommendation widget onto a monolithic CMS and you usually get batch-processed suggestions that are already stale by the time anyone sees them. That's barely different from no personalization at all.

What becomes possible once the data pipeline exists

A modernized, API-driven platform changes the equation because reader behavior, clicks, scroll depth, time on page, subscription status, can flow into a data pipeline in near real time instead of a nightly batch job. On AWS that typically means Kinesis or EventBridge handling event streaming, Amazon Personalize training recommendation models on actual reader behavior, and OpenSearch powering fast, relevant search and discovery. A homepage or app feed built this way reflects what a specific reader is engaging with right now, not what they clicked on yesterday.

Where generative AI fits in

Beyond recommendations, generative AI opens up a second kind of value: helping editorial teams instead of just ranking content for readers. Amazon Bedrock gives engineering teams access to foundation models for drafting article summaries, testing alternate headlines, auto-tagging content so it's easier to find, and powering conversational search where a reader can ask a question and get an answer pulled from the publisher's own archive. Done well, this doesn't replace editorial judgment. It clears out the repetitive work around it, so reporters and editors spend time on reporting and editing instead of manual tagging.

Personalization is a data problem before it is an AI problem

Worth saying plainly: publishers sometimes try to bolt AI personalization onto a platform that can't yet supply clean, real-time behavioral data. When that happens, the model was never the problem. The pipeline was. That's exactly why personalization belongs in the modernization conversation, not after it. The event streaming, the structured content, the decoupled APIs, that's the same architecture that makes AI-powered personalization possible in the first place.

Chapter 5

Architecture Patterns That Work

The reference pattern

A modernized publishing platform on AWS tends to follow the same basic pattern no matter the publisher's size. Content lives in a headless or hybrid CMS, backed by S3 for media and Aurora or DynamoDB for structured data. API Gateway and CloudFront handle delivery, with caching tuned by content type so breaking news invalidates fast while evergreen content stays cached longer. Reader behavior streams through Kinesis or EventBridge into a data layer feeding Amazon Personalize, and OpenSearch handles search and discovery. Compute runs on ECS, EKS, or Lambda depending on whether the workload is steady or event-driven.

Security and compliance are part of the architecture, not an afterthought

Publishing platforms carry subscriber data, payment details for paywalls, and often data that falls under GDPR or CCPA. IAM, encryption at rest and in transit, and WAF for public-facing endpoints need to be part of the architecture from day one, not something bolted on after a security review flags a gap. That's genuinely easier to get right on a modernized platform, since access controls and data flows are explicit in a decoupled system instead of buried somewhere in monolithic code.

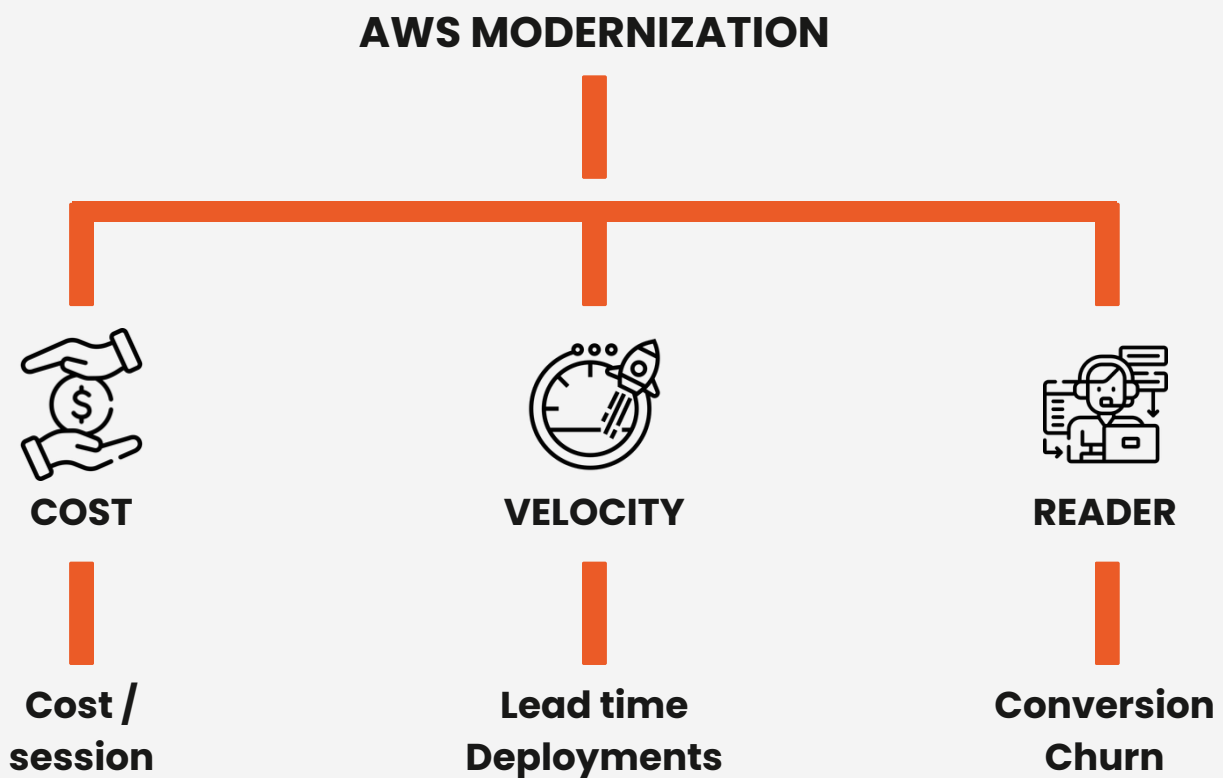
Built to extend, not just to launch

The real payoff of this architecture isn't that it solves today's requirements. It's that adding a new channel later, a smart TV app, a partner feed, a format that doesn't exist yet, means building against an API that's already there instead of re-architecting the whole platform again. That's the actual return on the investment, and it's easy to miss if you're only measuring the migration itself.

Chapter 6

Measuring What Matters: ROI and KPIs

The ROI Chain of Modernization



Infrastructure metrics

The most visible returns from AWS modernization show up in infrastructure economics: fixed-capacity spend turns into consumption-based pricing, spend drops during quiet traffic periods, and emergency scaling incidents mostly disappear. Track infrastructure cost per page view or per session over time. It's a cleaner efficiency signal than total cloud spend, since total spend naturally rises as traffic grows.

Engineering velocity metrics

Deployment frequency and lead time for changes tell you whether decoupling actually worked. A platform where a front-end change ships without touching the content layer, and vice versa, should show a real drop in time from feature request to production. Watch how long it takes a new engineer to become productive, too. A decoupled system with clear API boundaries is a lot easier to onboard into than a monolith carrying years of undocumented dependencies.

Business and reader metrics

The metrics that matter most to the business are the ones personalization and performance improvements are supposed to move: session duration, pages per session, subscription conversion, churn. These take longer to show clean results than infrastructure metrics do, so track them against a baseline set before migration starts. Leadership needs to see the trend line, not a single before-and-after snapshot that could be noise.

Chapter 7

Choosing the Right Modernization Partner

What actually matters in a partner

Most CMS modernization projects don't fail because of a bad architecture decision. They fail on execution: timelines nobody could realistically hit, a migration plan built around a single big-bang cutover, or a partner that hands over an architecture diagram and disappears before anyone has to run it. Look for AWS-specific engineering depth, real experience with publishing and media platforms specifically, and a migration approach built around incremental, low-risk delivery rather than one high-stakes launch date.

Why Forgeahead

Forgeahead works with engineering leaders to modernize legacy platforms on AWS and build the AI capabilities, from real-time personalization to generative AI-assisted editorial workflows, that legacy infrastructure was never built to support. The approach in this guide, phased migration, decoupled architecture, treating the data pipeline as the foundation rather than an afterthought, reflects how Forgeahead's engineering teams actually run these projects with publishing and media clients.

If your platform is the thing holding your team back, the conversation worth having isn't whether to modernize. It's where to start.

Get Started

Forgeahead helps engineering leaders at publishing and media companies modernize legacy CMS platforms on AWS and build the data infrastructure that makes AI-powered personalization possible. Reach out to the Forgeahead team to talk through what modernization could look like for your platform.

References

- Amazon S3. Amazon Web Services.
<https://aws.amazon.com/s3/>
- Amazon CloudFront. Amazon Web Services.
<https://aws.amazon.com/cloudfront/>
- AWS Lambda. Amazon Web Services.
<https://aws.amazon.com/lambda/>
- 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/>
- Amazon Aurora. Amazon Web Services.
<https://aws.amazon.com/rds/aurora/>
- Amazon DynamoDB. Amazon Web Services.
<https://aws.amazon.com/dynamodb/>
- Amazon Kinesis. Amazon Web Services.
<https://aws.amazon.com/kinesis/>
- Amazon EventBridge. Amazon Web Services.
<https://aws.amazon.com/eventbridge/>
- Amazon Personalize. Amazon Web Services.
<https://aws.amazon.com/personalize/>
- Amazon OpenSearch Service. Amazon Web Services.
<https://aws.amazon.com/opensearch-service/>
- Amazon Bedrock. Amazon Web Services.
<https://aws.amazon.com/bedrock/>
- AWS WAF. Amazon Web Services.
<https://aws.amazon.com/waf/>
- AWS Identity and Access Management (IAM). Amazon Web Services.
<https://aws.amazon.com/iam/>
- Amazon API Gateway. Amazon Web Services.
<https://aws.amazon.com/api-gateway/>



www.forgeahead.io



+1-650-948-1787