

TECH & AI CASE STUDY - CONSUMER LENDING - 2025

Kubernetes Microservices That Cut Deploy Time From Days to Minutes

Based on a delivery engagement with HarborCrest Lending

3 days to 10 min
DEPLOYMENT TIME

40x
DEPLOY FREQUENCY

99.5% to 99.95%
PLATFORM UPTIME

8 weeks to 5 days
ONBOARDING TO FIRST DEPLOY

A fillable practice edition: the full story, 8 actions,
6 reflections and a 90-day plan - work it with your team.

Free - no email required
emersontechnology.com

MAKE THIS PRACTICE BOOK YOURS

Work it, do not just read it

This practice book turns the HarborCrest Lending case study into a working session for your team. Read the full story first - what happened, what was built and what it changed. Then tick the 8 actions you can commit to, write short answers to the 6 reflections, and finish with a 90-day plan. Every field in this PDF is fillable: open it in Adobe Acrobat Reader, Chrome, Edge or macOS Preview, type directly into the boxes and save.

YOUR NAME

ROLE

COMPANY

TEAM / DEPARTMENT

DATE

EMAIL (OPTIONAL)

MY GOAL FOR THIS PRACTICE SESSION

- 01

The story - What happened at HarborCrest Lending - before, during and after the project.
- 02

The results - The numbers that changed, plus industry benchmarks from public research.
- 03

The 8-action checklist - Tick boxes as your organization completes each step. Owner and date fields included.
- 04

The 6 reflections - Fillable questions that pressure-test your own roadmap.
- 05

My 90-day plan - Three fillable phase cards that turn this case study into your roadmap.

RUN IT AS A 90-MINUTE TEAM SESSION

25 min READ Read the story together. Ask everyone to note the one number that surprised them most.	40 min PRACTICE Work the 8-action checklist and the 6 reflections. Assign an owner and a date to every action you commit to.	25 min COMMIT Fill in the 90-day plan. One goal per phase, one number that tells you it worked. Book a review date before you leave the room.
--	--	---

THE STORY

What happened, and why it worked

01 The Company

HarborCrest Lending is a 160-employee consumer lender that originates roughly \$400M in personal loans annually. Its product is a promise delivered at digital speed: borrowers apply online, receive decisions, upload documents, accept disclosures and manage repayments through HarborCrest's web platforms, while underwriting, servicing and payments run behind the scenes. In consumer lending, the software is the branch — there are no retail locations, so the quality of the platforms is the quality of the bank, and the speed of the software is the speed of the business.

The regulatory dimension makes this environment distinctive. Interest rates, disclosure requirements and servicing rules change on external schedules that HarborCrest does not control, and compliance deadlines are not negotiating positions — a rate-table change or a disclosure update must reach production by a date set by regulators, not by the release calendar. That combination of consumer expectations and regulatory hard deadlines means deployment capability is not an engineering vanity metric; it is a business constraint that determines whether the company can honor its obligations and compete for borrowers at the same time.

02 Where Things Stood

HarborCrest's entire operation — originations, servicing and underwriting — ran on a single 900,000-line Java monolith. The system worked, in the sense that loans were made and serviced, but its size had turned every change into a coordinated event. Every release, even a one-line disclosure fix, required a three-day deployment window, full regression testing and coordinated downtime across three teams. Because a single bad deployment could take down applications, servicing portals and payments simultaneously, deployment had become genuinely hazardous, and the organization responded the way organizations do to hazard: it avoided it. The company averaged one production deploy every six weeks.

The consequences compounded in every direction. Regulatory rate changes that competitors shipped in hours took HarborCrest weeks, forcing compliance and product teams into uncomfortable conversations about deadlines and risk. Product managers, unable to ship incrementally, batched features into risky quarterly mega-releases — each one a larger change set than the quarter before, which made each release weekend more dangerous and justified even longer gaps. Engineering attrition was climbing as top developers left for companies where shipping felt rewarding instead of hazardous, and the exit interviews were blunt about why. The monolith had become the company's operating ceiling: nothing about HarborCrest's product strategy could move faster than its deployment process.

03 The Challenge

The challenge was to dismantle a 900,000-line monolith that a regulated lender depends on, without pausing the business for two quarters. The monolith could not simply be frozen while it was rebuilt — originations had to keep flowing, servicing had to keep working, and the compliance calendar did not care that an architectural transition was underway. Google's DORA research reports that elite software performers deploy on demand, multiple times per day, while keeping change failure rates below 5%, and HarborCrest sat at the far opposite pole: deployments measured in months, failure rates shaped by fear. The gap was architectural, but closing it required a migration strategy that treated production stability as inviolable.

The problem decomposed into four interlocking difficulties. First, the seam problem: after years of growth, the monolith's internal boundaries did not correspond to business domains, so service boundaries had to be discovered and validated rather than assumed. Second, the data problem: shared database tables were the

monolith's hidden glue, and any decomposition that left shared tables behind would reproduce the monolith's deployment lockstep as a distributed system — the worst of both worlds. Third, the safety problem: deploying 14 services independently, 20 times a week, is only safe if rollback is automatic and regressions are caught by machines, not by customers. CNCF annual surveys have reportedly found around 96% of organizations using or evaluating Kubernetes, making it the de facto standard for container orchestration — but adopting Kubernetes was the easy part; organizing the organization around it was the hard part.

04 Discovery and Diagnosis

Emerson Technology began with two quarters of deliberate groundwork, starting with a domain-mapping exercise that combined code archaeology with workshop sessions across the engineering, compliance and operations teams. Version-control history revealed which parts of the monolith changed weekly — applications, decisioning, document generation — and which had barely shipped in a year, while incident history showed where failures concentrated. Overlaying the two maps produced a decomposition candidate list of 14 services aligned to business domains: applications, decisioning, servicing, payments and document generation among them. The team deliberately mapped boundaries to business domains rather than to existing team structures, because domain seams produce services that change independently and survive reorganizations.

The data diagnostic was the most consequential part of discovery. The team traced every shared table, every cross-module foreign key and every batch job that read another module's data, and produced the migration bill: each service would own its schema exclusively, and no shared table would survive the transition. The cost of that discipline is real — data migration work is slow, unglamorous and essential — and the team priced it into the plan rather than pretending the services could be carved around it. Discovery also instrumented the monolith with baseline metrics before anything moved: deployment frequency, lead time, change failure rate and time to restore, captured from the first sprint, so the migration's progress would be argued from numbers instead of narratives. Netflix's public engineering blog has long credited its microservices architecture with enabling rapid, independent deployments across hundreds of teams — a pattern now standard across streaming platforms — and the diagnosis closed by translating that pattern into HarborCrest's constraints: regulated data, compliance deadlines and zero tolerance for customer-visible regressions.

05 The Solution

We decomposed the monolith into 14 domain-aligned services over two quarters using the strangler-fig pattern, so releases continued throughout and every cutover was reversible.

Carving along domain seams

Services were extracted in value order, starting with the modules that changed most frequently and blocked the most work. Each new service runs as a Docker container on Amazon EKS, with Helm for templating and Argo CD for GitOps delivery — the desired state of production lives in git, and the cluster converges to it. Every service owns its database schema exclusively: no shared tables, no lockstep deploys, no hidden coupling through someone else's columns. Where modules needed data that lived elsewhere, the boundary was made explicit through APIs or events rather than through database reads. The strangler pattern meant the monolith kept serving traffic throughout, with each cutover routing a narrow slice of functionality to its new service and keeping the rollback path warm until the new owner had proven itself.

The delivery machinery

GitHub Actions builds and tests every change; Trivy security scans run on every image; and Argo Rollouts handles canary deployments with automatic rollback as the default path to production. A new version of a service takes a percentage of traffic first, error and latency signals are watched against explicit thresholds, and a breach

reverts the rollout without a human in the loop. Horizontal Pod Autoscaling absorbs origination spikes around marketing campaigns — scaled on submission-rate signals rather than CPU alone, because marketing-driven load patterns are business events, not infrastructure events. Kafka carries event streams between decisioning and servicing, replacing synchronous call chains that would otherwise couple the services' uptime and deployment calendars back together. This machinery is what makes 20 deploys a week rational: frequency is safe exactly to the degree that rollback is automatic.

Observability and safety

Prometheus and Grafana give every service its own dashboards, and OpenTelemetry traces follow a loan application across service boundaries, so an engineer can see where time goes instead of reconstructing it after an incident. Deploy-time feedback replaced post-incident archaeology: a canary that misbehaves is visible in minutes, not discovered in a support queue days later. Service ownership is documented, runbooks live with the code, and each service has a named owner — which turned the operations question "who do I call" from an archaeology project into an org-chart lookup. Together these layers made the safety case that compliance and leadership needed: the company was not trading stability for velocity, it was building the machinery that makes stability possible at higher velocity.

06 The First 90 Days

The first six weeks produced the decomposition plan in executable form: the 14 service definitions, the data-ownership map, the extraction order, and the delivery pipeline template that every new service would inherit. The first two services — applications and document generation — were extracted behind the strangler pattern, with shadow traffic validating behavior against the monolith before each cutover. The data migration for the applications service was the first real test of the schema-ownership discipline, and it ran cleanly, which mattered enormously for the team's confidence in the longer road ahead.

Weeks seven through ten extracted decisioning and servicing, the two domains at the heart of the lending operation, and introduced canary deployments with automatic rollback as the default path to production. The first automatic rollback occurred in week nine — a servicing change that tripped a latency threshold — and the event was treated as a milestone rather than a failure, because it demonstrated in production exactly the safety property the migration depended on. Weeks eleven and twelve completed payments extraction, wired Kafka event streams between decisioning and servicing, and ran the first onboarding experiment: a newly hired engineer took a guided first deploy on day five of employment, shipping a small change through the full pipeline. Baseline DORA metrics were reviewed with leadership at the end of the quarter, and the trajectory was already unmistakable — deployment frequency had moved from once every six weeks to multiple times per week, with no increase in customer-visible incidents. The monolith still existed at day 90, but it had stopped being the ceiling; it was now the residue, shrinking with every extraction.

07 The Results

Deploys that once consumed three-day release windows now complete in under ten minutes, and the team ships more than 20 times a week — a 40x frequency increase — with canary rollouts catching regressions before customers notice. Uptime climbed from 99.5% to 99.95% because service failures no longer cascade across the platform: a payments fault no longer takes down the application portal, and a servicing bug no longer blocks loan originations. Regulatory changes reach production in days rather than weeks, converting compliance deadlines from emergencies into ordinary work items. New engineers make their first production contribution within five days of joining instead of two months.

METRIC	BEFORE	AFTER
Deployment time	3 days	10 min
Deploy frequency	Baseline	40x
Platform uptime	99.5%	99.95%
Onboarding to first deploy	8 weeks	5 days

The organizational effects run deeper than the metrics. Product managers ship incrementally instead of batching mega-releases, so borrower feedback arrives weeks earlier and shapes what gets built next. Engineering attrition reversed as the daily experience of shipping changed — the exit-interview complaint about hazardous deployments disappeared, and recruiting conversations now lead with the delivery culture rather than apologizing for it. Release coordination, once a standing tax on the entire engineering budget, has largely evaporated; the hours that three teams once spent mobilizing for a release weekend now fund product work. And the compliance team, initially the program's most cautious stakeholders, became its advocates once regulatory changes started landing on deadline without release-weekend risk attached.

08 What Your Business Should Do

HarborCrest's path is the standard route out of monolith paralysis, and it works because of its sequence. Slice the monolith by domain: map service boundaries to business domains such as applications, decisioning and payments, not to existing team structures, because domain seams produce services that change independently and survive reorganizations. Use the strangler-fig pattern — route traffic incrementally to new services while the monolith keeps running — since big-bang rewrites freeze feature work for quarters while strangler migration lets the business ship throughout. Give every service its own data: assign each service a schema that only it can modify, because shared databases quietly re-couple microservices and reproduce the monolith's deployment lockstep within months, undoing the whole effort. Then automate canary deploys with rollback — canary a percentage of traffic to each new version and roll back automatically on error-budget breaches; automation makes frequent deploys safe enough to do daily.

Harden the platform for the load and the learning. Add autoscaling before the spike by tying horizontal pod autoscaling to business signals such as application submission rates, not just CPU utilization — marketing campaigns and rate changes create load patterns CPU alone never predicts. Stream events with Kafka rather than synchronous calls wherever workflows allow, because event-driven integration absorbs traffic bursts gracefully and lets consuming services upgrade independently without breaking producers. Measure DORA metrics from day one — deployment frequency, lead time, change failure rate and time to restore — since baselines turn debates about engineering velocity into numbers leadership can fund and reward. And invest in onboarding paths early: document service ownership, add runbooks and give new engineers a guided first deploy, because fast onboarding compounds, with every hire returning value in days rather than quarters. As you assess your own system, hold it against the questions this migration raised: how long a one-line fix really takes to reach production today, which parts of your monolith change weekly and which have not shipped in a year, what your cadence would be if every deploy carried zero fear of taking the platform down, and whether a new engineer could safely ship in their first week. Work through these eight steps with your team using the action checklist in this practice book, and revisit the reflection questions each quarter so the architecture keeps serving the business instead of becoming its new constraint.

REAL-WORLD PROOF & BENCHMARKS

Google's DORA research reports that elite software performers deploy on demand, multiple times per day, while keeping change failure rates below 5%. - DORA Report, 2021

CNCF annual surveys have reportedly found around 96% of organizations using or evaluating Kubernetes, making it the de facto standard for container orchestration. - CNCF, 2021

Netflix's public engineering blog has long credited its microservices architecture with enabling rapid, independent deployments across hundreds of teams — a pattern now standard across streaming platforms. - Netflix Tech Blog

THE PRACTICE PAGES

The 8-action checklist

Work through the list with your team. Tick each box as your organization completes the step, and fill in an owner and a target date so every action has a name and a deadline.

☐

1. Slice the monolith by domain

Map service boundaries to business domains such as applications, decisioning and payments, not to existing team structures. Domain seams produce services that change independently and survive reorganizations.

OWNER

TARGET DATE

STATUS

--	--	--

☐

2. Use the strangler-fig pattern

Route traffic incrementally to new services while the monolith keeps running. Big-bang rewrites freeze feature work for quarters; strangler migration lets the business ship throughout.

OWNER

TARGET DATE

STATUS

--	--	--

☐

3. Give every service its own data

Assign each service a schema that only it can modify. Shared databases quietly re-couple microservices and reproduce the monolith's deployment lockstep within months, undoing the whole effort.

OWNER

TARGET DATE

STATUS

--	--	--

☐

4. Automate canary deploys with rollback

Canary a percentage of traffic to each new version and roll back automatically on error-budget breaches. Automation makes frequent deploys safe enough to do daily.

OWNER

TARGET DATE

STATUS

--	--	--

☐

5. Add autoscaling before the spike

Tie horizontal pod autoscaling to business signals such as application submission rates, not just CPU utilization. Marketing campaigns and rate changes create load patterns CPU alone never predicts.

OWNER

TARGET DATE

STATUS

--	--	--

**6. Stream events with Kafka, not calls**

Replace synchronous service-to-service call chains with event streams wherever workflows allow. Event-driven integration absorbs traffic bursts gracefully and lets consuming services upgrade independently without breaking producers.

OWNER

TARGET DATE

STATUS

--	--	--

**7. Measure DORA metrics from day one**

Track deployment frequency, lead time, change failure rate and time to restore from the first sprint. Baselines turn debates about engineering velocity into numbers leadership can fund and reward.

OWNER

TARGET DATE

STATUS

--	--	--

**8. Invest in onboarding paths early**

Document service ownership, add runbooks and give new engineers a guided first deploy. Fast onboarding compounds: every hire starts returning value within days, not quarters.

OWNER

TARGET DATE

STATUS

--	--	--

AFTER THE CHECKLIST: OUR TOP 3 PRIORITIES

From the eight actions above, write down the three you will start first - and the single number you expect each one to move. Three priorities with owners beat eight without any.

PRIORITY 1	PRIORITY 2	PRIORITY 3
NUMBER IT SHOULD MOVE	NUMBER IT SHOULD MOVE	NUMBER IT SHOULD MOVE

THE PRACTICE PAGES

Reflections for your team

Use these questions in your next planning session and type short answers into the boxes. The gaps you find are usually the fastest wins.

Q1. How long does a one-line fix take to reach production in your organization today — honestly?

Q2. Which parts of your monolith change weekly, and which have not shipped in a year?

Q3. What would your release cadence be if every deploy carried zero fear of taking the platform down?

Q4. How much of your engineering budget funds release coordination rather than customer-facing improvements?

Q5. Could a new engineer safely ship to production in their first week — and if not, what blocks them?

Q6. If a regulator required a change next Monday, which systems could you update in time?

THE PRACTICE PAGES

My 90-day plan

Finish the session by committing to a plan. Three phases, one goal each, one number that tells you it worked. Bring this page to your next leadership review - or to a free consultation with Emerson Technology, and we will pressure-test it with you.

Days 1-30 Focus: audit and alignment

PRIMARY GOAL**KEY ACTIONS****HOW WE MEASURE SUCCESS****OWNER**

Days 31-60 Focus: build and prove

PRIMARY GOAL**KEY ACTIONS****HOW WE MEASURE SUCCESS****OWNER**

Days 61-90 Focus: scale and measure

PRIMARY GOAL

KEY ACTIONS

HOW WE MEASURE SUCCESS

OWNER

Tip: save a copy per team so each group keeps its own answers. Revisit the reflections every quarter - the questions stay the same even as the numbers move.

BOOK THE REVIEW BEFORE YOU CLOSE THE BOOK

Plans without a review date drift. Before you close this practice book, agree on when you will come back to these pages, who will run that session, and when you will check the numbers again. If you want an outside pair of eyes, bring this page to a free consultation with Emerson Technology and we will pressure-test the plan with you.

REVIEW DATE	REVIEWED BY	NEXT CHECK-IN
------------------------------------	------------------------------------	--------------------------------------

Ready for a transformation like HarborCrest Lending's?

Emerson Technology designs and builds the data, AI, cloud and automation systems behind results like these. On a free consultation call we will map this exact playbook to your business - your data, your constraints, your numbers.

Book a free consultation

support@emersontechnology.com

or start online: emersontechnology.com/contact