Building PresCoach: An AI-DLC Retrospective

How a CTO and an AI Development Companion Built a Production-Grade Agentic System in Five Days

---

Executive Summary

The Presentation Coaching Platform (PresCoach) is a meaningfully complex, end-to-end agentic system that ingests audio presentations, processes them through vector embeddings, and prepares them for multi-dimensional AI coaching evaluation. It was built collaboratively between a CTO and Kiro — an AI-powered development environment — in approximately 8 working hours spread across less than two days of elapsed time. The remaining time was occupied by other work commitments.

This document tells the story of that build: the architectural decisions, the development process, and how Agentic Development Life Cycle (AI-DLC) practices enabled the rapid creation of a system comprising a frontend SPA, serverless backend, event-driven processing pipeline, multi-model AI integration, and full CI/CD automation — all built on AWS-native services that scale from zero to high-volume concurrency with no infrastructure to manage.

The Starting Point

I began with a clear vision and a written specification. The initial document described the Presentation Coaching Platform's business requirements, functional and non-functional requirements, and architectural guidance. It laid out a system that would:

- Accept audio and video presentations from users

- Process them through vector embeddings for semantic understanding
- Route them to a set of independent AI evaluation agents
- Produce comprehensive coaching reports

This specification became the foundation for everything that followed. Kiro ingested it and together we decomposed the system into **Work Units** — the fundamental building blocks of the AI Development Life Cycle.

Work Unit Decomposition

Using AI-DLC methodology, PresCoach was decomposed into four Work Units, each with clearly defined boundaries, interfaces, and responsibilities:

Work Unit	Responsibility	Status
Frontend SPA	User interface, authentication, file upload UX	✅ Implemented
Upload & Storage	File reception, validation, metadata persistence, event publishing	✅ Implemented
Preparation Workflow	Format validation, audio extraction, chunking, embedding, vector storage, handoff	✅ Implemented
Agentic Evaluation	Multi-agent coaching evaluation, report generation	📋 Specified

Each Work Unit functions as an independent **Bolt** — a self-contained, deployable unit of capability that communicates with other Bolts exclusively through well-defined interfaces (SQS queues, S3 events, API contracts). This Bolt-based architecture means each Work Unit can be developed, tested, deployed, and scaled independently.

Work Unit 1: The Frontend SPA

What It Is

A single-page application that provides users with a professional, responsive interface for uploading presentations and tracking their processing status. It's served globally through CloudFront with sub-second load times.

Technical Profile

Aspect	Choice
Architecture	Vanilla JavaScript SPA with client-side routing
Hosting	S3 static hosting + CloudFront CDN
Authentication	AWS Cognito (OAuth 2.0 Authorization Code Grant with PKCE)
Styling	Custom CSS with responsive design
Testing	Vitest with property-based tests (fast-check)

How It Was Built

Kiro generated the complete frontend from the spec: semantic HTML, modular JavaScript (router, auth module, API client, view components), and responsive CSS. I reviewed the authentication flow — the PKCE implementation was particularly important to get right since the SPA is a public client that cannot store secrets.

The authentication architecture uses Cognito's hosted UI for login/signup, stores tokens in memory (not localStorage — a security best practice for SPAs), and handles token refresh transparently. Users never see an expired session unless their refresh token has also expired.

Key Design Decisions

- **No framework (React/Vue/Angular):** For an MVP with two views (Upload and List), a framework adds bundle size and complexity without meaningful benefit. The vanilla approach loads in milliseconds and has zero build step for static files.
- **Cognito Hosted UI:** Building custom login forms (which must handle MFA, password policies, and verification flows) is never an acceptable option. The hosted UI provides production-grade Au/Az in minutes.
- **CloudFront + S3:** Global CDN delivery with zero servers. The entire frontend costs pennies per month at MVP scale.

Work Unit 2: Upload & Storage

What It Is

The backend service that receives presentation files, validates them, stores them in S3, persists metadata in DynamoDB, and publishes events to trigger downstream processing. It's the bridge between the user-facing webapp and the processing pipeline.

Technical Profile

Aspect	Choice
API	AWS API Gateway (HTTP API) with JWT authorizer
Compute	AWS Lambda (Python 3.12)
Storage	S3 (files) + DynamoDB (metadata)
Messaging	SQS (processing trigger) + SNS (error notifications)
Auth	Cognito JWT validation at the API Gateway level
IaC	AWS CDK (Python)

Aspect	Choice
Data modeling	Pydantic v2
Testing	pytest + Hypothesis (property-based) + moto (AWS mocking)

How It Integrates with the Webapp

```
Webapp → API Gateway (JWT auth) → Lambda → S3 (presigned URL) → DynamoDB
                                                    ↓
                                              S3 Event → Lambda → SQS
```

1. The webapp calls `POST /submissions` with metadata and receives a presigned S3 PUT URL
2. The webapp uploads the file directly to S3 using the presigned URL (no file passes through Lambda)
3. S3 fires a PutObject event notification to the `confirm_upload` Lambda
4. That Lambda validates the upload, updates DynamoDB status, and publishes an SQS message

This presigned URL pattern is critical: it means Lambda never touches the file bytes. A 500 MB video upload goes directly to S3 at wire speed, while Lambda handles only the ~1 KB metadata request. This keeps Lambda execution under 1 second regardless of file size.

Serverless and Native Service Choices

Every component is a fully managed AWS-native service:

Component	Why This Service
API Gateway (HTTP API)	Native JWT auth, automatic scaling, pay-per-request, no servers
Lambda	Millisecond billing, scales to thousands of concurrent executions, zero idle cost

Component	Why This Service
S3	Unlimited storage, 11 9s durability, event notifications built-in
DynamoDB	Single-digit millisecond latency, on-demand scaling, no capacity planning
SQS	Decouples upload from processing, built-in retry, DLQ for failures
Cognito	Managed auth with MFA, password policies, token management — no auth server

Scalability and Cost

Scale from zero: When no one is using the system, the cost is effectively $0. No EC2 instances running, no ECS tasks, no RDS instances. API Gateway, Lambda, DynamoDB (on-demand), and SQS all charge only for actual usage.

Scale to high volume: API Gateway and Lambda handle thousands of concurrent requests without any configuration change. DynamoDB on-demand mode auto-scales read/write capacity. SQS is virtually unlimited in throughput. The system handles 1 upload/day and 10,000 uploads/hour with the same code and infrastructure.

Cost at scale: At 1,000 uploads/month with 50 MB average file size:

- Lambda: ~$0.05 (50K ms of execution)
- API Gateway: ~$0.35
- S3: ~$1.15 (50 GB storage + requests)
- DynamoDB: ~$0.15
- **Total: ~$2/month** for a production backend

Work Unit 3: Preparation Workflow

What It Is

An event-driven processing pipeline that validates uploaded files, optionally extracts audio from video, divides audio into chunks, creates vector embeddings using Amazon Bedrock, stores embeddings in a vector store, and hands off results to the Agentic Evaluation Work Unit. It's implemented as an AWS Step Functions Standard Workflow orchestrating nine Lambda functions.

Technical Profile

Aspect	Choice
Orchestration	AWS Step Functions (Standard Workflow)
Compute	9 AWS Lambda functions (Python 3.12)
AI/ML	Amazon Bedrock (Nova Multimodal Embeddings)
Audio processing	AWS Elemental MediaConvert
Vector storage	S3 (configurable for OpenSearch Serverless)
Configuration	AWS Systems Manager Parameter Store
Event trigger	EventBridge Pipe (SQS → Step Functions)
Messaging	SQS FIFO (handoff) + SNS (errors)
Testing	203 tests (22 property, 139 unit, 6 integration, 36 CDK)

The Processing Pipeline (Happy Path)

```
SQS Message → EventBridge Pipe → Step Functions
    ↓
LoadConfig (SSM) → ParseMessage → UpdateStatus("Processing")
    ↓
ValidateFormat → CheckVideoFlag
    ↓ (audio)
ChunkAudio → CreateEmbeddings (Map, 10x concurrent) → StoreVectors
```

```
    ↓
PublishHandoff → UpdateStatus("Completed")
```

How Kiro Built This

This was the most complex Work Unit. Kiro:

1. Generated the complete requirements document with 11 requirements and detailed acceptance criteria
2. Designed the architecture including the Step Functions ASL (Amazon States Language) definition
3. Produced a task plan with 46 implementation tasks organized in a dependency graph
4. Executed all 46 tasks, running them in parallel waves based on the dependency DAG
5. Wrote and validated 203 tests — including 10 property-based tests that verify correctness invariants across randomly generated inputs
6. Generated the CDK infrastructure-as-code
7. Produced a CloudFormation template for direct deployment

The property-based testing approach is particularly noteworthy. Rather than testing specific examples, Kiro defined formal correctness properties (e.g., "for any audio duration, chunk size, and overlap, the chunking algorithm shall produce chunks where the first chunk starts at 0, the last chunk covers the end, and no content is skipped") and used the Hypothesis library to verify these properties hold across hundreds of randomly generated test cases.

Serverless and Native Service Choices

Component	Why This Service
Step Functions (Standard)	Long-running orchestration (embeddings may take >5 min), built-in retry with backoff/jitter, execution history, native error handling
EventBridge Pipes	Zero-code SQS → Step Functions integration, no Lambda trigger needed
Lambda (10 functions)	Each function does one thing, independently scalable, millisecond billing

Component	Why This Service
Bedrock (Nova Embeddings)	Serverless AI inference, pay-per-request, no model hosting
MediaConvert	Serverless transcoding, pay-per-minute of output, no FFmpeg servers
SSM Parameter Store	Runtime reconfiguration without redeployment, free tier covers it
S3 (vector store)	Unlimited, durable, cheap — suitable for MVP vector storage
SQS FIFO (handoff)	Preserves per-submission ordering, exactly-once delivery, loose coupling

Scalability and Cost

Scale from zero: When idle, the only recurring cost is S3 storage for existing embeddings and SSM parameter storage (free tier). Step Functions, Lambda, Bedrock, and MediaConvert all have zero idle cost.

Scale to high volume: The Step Functions Map state processes embedding chunks with 10x concurrency by default (configurable up to 40). Lambda scales independently per function. Bedrock handles concurrent inference requests. Processing 100 presentations simultaneously requires zero configuration changes.

Cost at scale (1,000 presentations/month, 30-minute average audio):

- Step Functions: ~$2.50 (state transitions across 1,000 executions)
- Lambda: ~$8.00 (total execution across all functions — chunking and embedding are the heaviest)
- Bedrock embeddings: ~$350.00 (each 30-min presentation produces ~72 chunks at 30s/5s overlap; 72,000 embedding calls/month)
- MediaConvert: $0 (audio-only, no video processing in MVP)
- S3 vector storage: ~$1.50 (72,000 JSON files/month, ~2.4 GB cumulative)

- **Total: ~$362/month** for processing 1,000 presentations This is about $0.36 per 30 minute presentation. We will add in the Agentic analysis.

Retry Logic and Error Handling

Every Task state in the Step Function has automatic retry:

- **Exponential backoff** with configurable initial intervals (1s for DynamoDB, 2s for Lambda, 5s for Bedrock, 30s for MediaConvert)
- **Backoff rate** of 2x per retry
- **3 max attempts** before routing to the HandleFailure state

The HandleFailure state performs three actions atomically:

1. Updates DynamoDB status to `Failed`
2. Publishes an SNS error notification (best-effort — won't fail the workflow)
3. Routes the original message to the appropriate Dead Letter Queue

---

Multi-Region Considerations

Current Architecture (Single Region)

PresCoach currently deploys to `us-east-1`. All resources — API Gateway, Lambda, DynamoDB, S3, Step Functions, SQS, Cognito — reside in a single region.

Current resilience:

- S3: 11 9s durability (data replicated across 3+ AZs within the region)
- DynamoDB: Synchronous replication across 3 AZs
- Lambda: Automatically runs across multiple AZs
- SQS: Replicated across AZs within the region
- Step Functions: Regionally resilient (multi-AZ)

The system is resilient to individual AZ failures with no configuration. Only a complete regional outage would take the system offline.

Path to Multi-Region

The architecture was explicitly designed for multi-region expansion without re-architecture:

Component	Multi-Region Upgrade
DynamoDB	Enable Global Tables (one checkbox in console) — active-active replication
S3	Enable Cross-Region Replication (CRR) to a secondary bucket
Cognito	Currently single-region — would need a separate User Pool per region with federation
API Gateway	Deploy to second region + Route 53 latency-based routing
Lambda	Deploy same code to second region (CI/CD pipeline adds a target)
Step Functions	Deploy same state machine to second region
SQS	Create queues in second region — producers route based on region affinity
CloudFront	Already global — just add the new regional API Gateway as an origin

The Bolt architecture makes this straightforward: each Work Unit is already self-contained. Deploying a second region means running the same CDK/CloudFormation stack in `us-west-2` and configuring cross-region data replication. No application code changes required.

Estimated effort: 1-2 days for active-passive failover, 1-2 weeks for active-active with conflict resolution.

Deployment and CI/CD

How It Was Deployed

The initial deployment was done directly from the command line (CloudShell and local terminal):

- **Upload Service:** Deployed via `cdk deploy` from the `upload-service/cdk/` directory
- **Frontend:** Deployed via `aws s3 sync` + CloudFront invalidation
- **Preparation Workflow:** Deployed via `aws cloudformation create-stack` using a CloudFormation template

CI/CD Implementation

Kiro created three CodePipeline definitions under `cicd/`:

Webapp & Upload Service Pipelines (`cicd/webapp-upload/`):

Pipeline	What It Does
`prescoach-dev-kiro-webapp`	Syncs webapp/ to S3, invalidates CloudFront
`prescoach-dev-kiro-upload-service`	Runs CDK deploy for the upload-service stack
`prescoach-dev-kiro-webapp-upload-full-deploy`	Upload Service first, then Webapp

Preparation Workflow Pipelines (`cicd/preparation-workflow/`):

Pipeline	What It Does
`prescoach-dev-kiro-prep-workflow-test`	Runs 203 tests (property + unit + integration)
`prescoach-dev-kiro-prep-workflow-deploy`	CDK deploy for Step Functions + Lambda + infrastructure

Pipeline	What It Does
`prescoach-dev-kiro-prep-workflow-full-deploy`	Tests gate the deploy — broken code never reaches production

All pipelines source from GitHub (main branch) and can be triggered manually from CLI or the AWS Console. The test-then-deploy pattern ensures the system's correctness properties are verified before every infrastructure change.

The AI-DLC Process

How Kiro Worked

The collaboration followed a structured Agentic Development Life Cycle:

1. **I provided the vision and constraints.** Business requirements, architectural guidance, non-functional requirements, security posture.
2. **Kiro decomposed into Work Units.** Each unit got a formal requirements document, a technical design document, and a task plan with dependency graphs.
3. **Kiro executed in waves.** Tasks within each Work Unit were organized into dependency-ordered waves and executed in parallel where possible. The preparation workflow's 46 tasks were completed in 14 waves.
4. **Property-based testing validated correctness.** Rather than relying solely on example-based tests, Kiro defined formal correctness properties and verified them across hundreds of randomly generated inputs. This catches edge cases that manual testing misses.
5. **I made decisions at branch points.** Architecture choices (serverless vs. containers, S3 vs. OpenSearch for vectors, Standard vs. Express workflow), feature scope (video disabled for MVP), and naming conventions.
6. **Kiro handled the volume.** 203 tests, 10 Lambda functions, a 15-state Step Functions ASL definition, 10 SSM parameters, complete IAM policies with least-privilege, CDK infrastructure, CloudFormation templates, CI/CD pipelines, and installation documentation.

What Made This Fast

- **Spec-driven development:** Writing requirements before code means no wasted implementation effort.
- **Wave-based parallelism:** Independent tasks execute concurrently, compressing build time.
- **Property-based testing:** Instead of writing dozens of example tests, define the property once and let the machine generate test cases.
- **AWS-native services:** No time spent configuring servers, containers, load balancers, or databases. Every service is API-call-away from working.
- **Bolt architecture:** Each Work Unit is isolated. The preparation workflow was built without touching any upload-service code.

What's Next

The Agentic Evaluation Work Unit — the intelligence layer — is fully specified and ready for implementation. It will use Amazon Bedrock AgentCore to orchestrate multiple independent evaluation agents (delivery analysis, structure assessment, pacing evaluation, executive presence scoring, and more) coordinated by a Coaching Supervisor agent that reasons about which evaluations are appropriate for each presentation.

The Bolt architecture means this Work Unit can be built, tested, and deployed without touching anything that already works. The handoff contract (the SQS FIFO message from the Preparation Workflow) is already defined and operational.

Summary

Metric	Value
Working hours	~40 hours

Metric	Value
Elapsed time	<7 days
Work Units	4
Lambda functions	9 (preparation workflow) + 3 (upload service)
Tests written	203+
Property-based tests	10 formal correctness properties
AWS services used	14 (Lambda, Step Functions, API Gateway, S3, DynamoDB, SQS, SNS, Cognito, CloudFront, EventBridge Pipes, SSM, Bedrock, MediaConvert, CloudWatch)
Servers managed	0
Idle cost	~$0
CI/CD pipelines	6

PresCoach demonstrates that a meaningfully complex, production-grade agentic system can be built rapidly when you combine clear architectural vision with AI-DLC practices and AWS-native serverless services. The result is a system that scales from a demo to thousands of users with no re-architecture — and it costs nothing when no one's using it.

Cost Analysis: Why This Architecture Pays for Itself

This section provides a detailed cost breakdown for PresCoach, with particular attention to the decisions that make costs predictable at low scale and manageable at high scale — especially for workloads with extreme variability (imagine usage spiking 50x before re:Invent or an AWS Summit, then dropping back to near-zero the following week).

The Zero-Idle Principle

Every service in PresCoach was chosen because it charges **nothing when idle**. There are no minimum fees, no reserved capacity, and no always-on infrastructure:

Service	Idle Cost	Charged For
Lambda	$0	Invocations + duration
API Gateway	$0	Requests
Step Functions	$0	State transitions
DynamoDB (on-demand)	$0	Read/write capacity units consumed
SQS	$0	Requests
SNS	$0	Publishes + deliveries
Bedrock	$0	Input/output tokens
MediaConvert	$0	Minutes of output
EventBridge Pipes	$0	Invocations
SSM Parameter Store	$0	Standard parameters (storage + reads) are free
S3	Storage only	$0.023/GB/month (only for data already stored)
CloudFront	$0 (nearly)	Requests + data transfer
Cognito	$0	First 50,000 MAU free

Result: A fully deployed PresCoach environment with no active users costs approximately **$0.05/month** (S3 storage for the webapp files + a few KB of CloudWatch logs).

SSM Parameter Store: Free Configuration Management

The Preparation Workflow reads 10 configuration parameters from SSM Parameter Store at the start of every execution. This is a deliberate architectural choice: it enables runtime reconfiguration (swap embedding models, change chunk sizes, toggle features) without redeploying code.

Cost: $0. Standard parameters are free for both storage and API calls at standard throughput (up to 40 transactions per second). You would need 40+ simultaneous workflow executions starting in the same second to approach the throughput limit — at which point higher throughput mode costs $0.05 per 10,000 interactions.

Even at 1 million workflow executions per month, SSM reads cost nothing at standard throughput. The 10 parameters are fetched via a single `GetParametersByPath` call (not 10 individual calls), making this extremely efficient.

S3 as Vector Store: The MVP Decision

The choice to store vector embeddings in S3 (rather than a dedicated vector database like OpenSearch Serverless or Pinecone) was a deliberate cost-optimization decision for the MVP phase.

How it works: Each embedding is stored as a JSON file at `{submission_id}/embeddings/chunk_{NNNN}.json` containing the embedding vector and metadata. The Agentic Evaluation agents retrieve embeddings by listing and reading these files when processing a submission.

Why S3 for MVP:

Factor	S3	OpenSearch Serverless
Idle cost	$0 (storage-only)	~$175/month minimum (2 OCUs)
Storage cost	$0.023/GB/month	$0.24/GB/month (10x more)

Factor	S3	OpenSearch Serverless
Read latency	~50-100ms per file	~10ms per query
Similarity search	Not supported (must load all)	Native vector similarity
Setup complexity	Zero (bucket already exists)	Significant (collection, index, policies)

The cost reality for variable workloads:

Imagine PresCoach usage before re:Invent:

- **Normal month:** 100 presentations → ~300 embedding files → ~1 MB storage → $0.000023/month
- **re:Invent prep month:** 10,000 presentations → ~30,000 embedding files → ~100 MB storage → $0.0023/month
- **Post-conference month:** 50 presentations → near zero incremental cost

With OpenSearch Serverless, you'd pay the $175/month minimum **every month**, including the quiet ones. Over a year with 2 busy months and 10 quiet months, that's $2,100 for a vector database vs ~$0.03 for S3.

When to upgrade: Move to OpenSearch Serverless (or Amazon Aurora with pgvector) when:

- You need real-time similarity search across submissions (not just per-submission retrieval)
- Read latency of 50-100ms per embedding file becomes a bottleneck
- You're processing >50,000 presentations/month consistently

The SSM parameter `vector-store-type` and `vector-store-endpoint` make this a configuration change, not a code change.

S3 Lifecycle Policies: Controlling Storage Costs Long-Term

S3 storage is cheap, but it's not free — and without lifecycle management, costs grow indefinitely as presentations accumulate. PresCoach uses lifecycle policies across all buckets to keep storage costs bounded.

The key insight: Users already have their audio/video files locally. Once processing is complete, there's no reason to keep source media in hot storage. Embeddings are intermediate artifacts — they're only needed during evaluation. Reports, however, need to be accessible long-term.

Bucket	Content	Lifecycle Policy
`prescoach-dev-kiro-uploads`	Original audio/video files + processed chunks	**30 days:** Transition to S3 Infrequent Access → **90 days:** Transition to Glacier Instant Retrieval → **365 days:** Delete
`prescoach-dev-vectors-*`	Embedding JSON files	**90 days:** Transition to S3 Infrequent Access → **180 days:** Delete (embeddings are recreatable from source audio)
Reports bucket (future)	Generated PDF coaching reports	**90 days:** Transition to S3 Infrequent Access → **Never delete** (reports are the deliverable users paid for)

Why this matters at scale:

Without lifecycle policies (1,000 presentations/month × 30-min audio × 12 months):

- Uploads bucket: ~600 GB accumulated → $13.80/month in Standard storage
- Vectors bucket: ~29 GB accumulated → $0.67/month

With lifecycle policies:

- Uploads bucket: ~50 GB in Standard (current month) + older data in Glacier ($0.004/GB) → ~$3.50/month
- Vectors bucket: ~4.8 GB in Standard (last 2 months) + older deleted → ~$0.11/month

Annual savings: ~$120/year at 1,000 presentations/month. At higher volumes, the savings compound significantly.

The lifecycle policy on the vector store is particularly aggressive (delete after 180 days) because embeddings can always be regenerated from the source audio if needed. The `LoadConfig` Lambda reads the model version from SSM, so re-embedding with an upgraded model is a routine operation anyway — old embeddings become stale as models improve.

Cost Scenarios: From Demo to Conference Spike

Scenario 1: Demo / PoC (10 presentations/month)

Service	Cost
Lambda (all functions)	$0.01
Step Functions	$0.01
Bedrock embeddings	$0.50
S3 (storage + requests)	$0.01
API Gateway	$0.01
DynamoDB	$0.01
Everything else	$0.00
Total	**~$0.55/month**

Scenario 2: Regular Use (500 presentations/month)

Service	Cost
Lambda	$0.25
Step Functions	$0.13
Bedrock embeddings	$25.00
S3 (storage + requests)	$0.50
API Gateway	$0.50

Service	Cost
DynamoDB	$0.25
CloudFront	$1.00
Total	**~$28/month**

Scenario 3: Conference Spike (10,000 presentations in one week)

Service	Cost (that week)
Lambda	$5.00
Step Functions	$2.50
Bedrock embeddings	$500.00
S3 (requests)	$5.00
API Gateway	$3.50
DynamoDB	$5.00
CloudFront	$10.00
Total	**~$531 for the spike week**

Then it drops back to Scenario 1 the following week. No capacity to de-provision, no instances to scale down, no waste.

Scenario 4: The following quiet month

Service	Cost
S3 (stored data from spike)	$2.30

Service	Cost
Everything else	$0.05
Total	**~$2.35/month**

The Bedrock Cost Dominance

At any meaningful scale, Bedrock embedding inference dominates the bill (~90% of costs in Scenario 3). This is by design — the most expensive operation is the one that provides the most value (semantic understanding of presentation audio).

Cost optimization levers for Bedrock:

- **Batch processing** (configurable via SSM): Group chunks for batch inference when supported, reducing per-invocation overhead
- **Model selection** (configurable via SSM): Switch to a cheaper model for less critical use cases
- **Chunk size tuning** (configurable via SSM): Larger chunks = fewer embedding calls per presentation

All of these are runtime configuration changes — no redeployment needed.

Why Not Containers?

A common question: "Why not ECS/Fargate or EKS?"

For PresCoach's workload profile (variable, spiky, often idle):

Factor	Serverless (current)	Containers (ECS Fargate)
Idle cost	$0	~$30-100/month (minimum tasks)
Scale-to-zero time	Instant	N/A (must keep min tasks running)
Scale-up time	~100ms (Lambda cold start)	30-60s (task provisioning)

Factor	Serverless (current)	Containers (ECS Fargate)
Conference spike handling	Automatic, instant	Requires auto-scaling config + warm-up
Operational overhead	Zero	ALB, task definitions, health checks, ECR

Containers make sense when you have sustained high throughput (>1000 req/s consistently) where Lambda's per-invocation pricing exceeds Fargate's per-hour pricing. PresCoach won't hit that threshold for years, if ever.

Total Cost of Ownership: Year One

Assuming the usage pattern described (2 busy months, 10 quiet months):

Component	Annual Cost	
2 busy months (5,000 presentations each)	~$130	
10 quiet months (100 presentations each)	~$55	
S3 storage (accumulated)	~$5	
CloudFront (CDN)	~$15	
Cognito (under 50K MAU)	$0	
SSM Parameter Store	$0	
CI/CD pipelines	$12 (CodeBuild minutes)	
	Year 1 Total	**$217**

Compare this to a traditional architecture (EC2 + RDS + ElastiCache + ALB): minimum **~$3,600/year** ($300/month) even when idle. The serverless approach saves over $3,000 in year

one for a variable workload.