TeamMatch System Architecture (AWS Serverless + Agent Microservice)

1. Overview

TeamMatch is a cloud-native web application designed to form balanced student teams in project-based computer science courses. The architecture separates fast user-facing operations from computationally intensive matching workflows. A lightweight serverless API handles user interaction, validation, and persistence, while a dedicated Agent Microservice performs long-running optimization tasks asynchronously.

This separation ensures that instructors can trigger team generation without waiting for the full optimization process to complete. The system mirrors real production architectures where request-response traffic is isolated from background workflows.

2. Architectural Design Principles

Separation of Concerns
Frontend handles presentation. API handles validation and routing. Agent handles computation. Database stores truth.

Asynchronous Processing
Matching is treated as a job, not an API request.

Deterministic Optimization
Identical inputs produce identical outputs. Tie-breaking and sorting are consistent.

Explainability First
Every team output includes scoring metrics and natural-language reasoning.

Cloud-Native Scalability
API auto-scales. Agent scales based on queue depth. Infrastructure is managed.

Security by Design
Least-privilege IAM roles. Secrets stored in AWS Secrets Manager.

Operational Realism
Uses SQS, Step Functions, ECS Fargate, and proper workflow orchestration.

3. High-Level Architecture

Students and instructors interact with a React/Next.js frontend hosted on **AWS S3** and delivered through **CloudFront**.

The frontend communicates with **AWS API Gateway**, which routes requests to **AWS Lambda** functions.

When an instructor triggers "Run Match":

1. Lambda validates input.
2. A MatchRun record is created in DynamoDB.
3. A job message is published to **AWS SQS**.
4. **AWS Step Functions** orchestrates the workflow.
5. The workflow invokes the **Agent Microservice** running on **AWS ECS Fargate**.
6. The agent performs LLM weighting, optimization, scoring, and explainability.
7. Results are written back to **DynamoDB**.
8. Job status is updated and surfaced to the UI.

4. Frontend Layer (React / Next.js on S3 + CloudFront)

The frontend is a static web build deployed to S3 and globally distributed via CloudFront.

Responsibilities

- Student survey submission (skills, availability, experience, leadership preference)
- Instructor configuration of constraints
- Triggering match runs
- Displaying teams and scoring breakdowns
- Polling job status

Example Frontend Functions

submitStudentProfile(courseId, studentId, payload)
updateCourseConstraints(courseId, payload)
startMatchRun(courseId, runConfig)
getRunStatus(runId)
getTeams(courseId, runId)

The frontend contains no optimization logic.

5. Backend API Layer (API Gateway + Lambda)

API Gateway exposes REST endpoints. Lambda functions serve as the stateless execution layer.

Responsibilities

- Input validation and sanitization
- Role-based authorization
- DynamoDB persistence
- Publishing SQS job messages
- Returning structured JSON responses

Example Endpoints

POST /courses/{courseId}/students/{studentId}/profile
PUT /courses/{courseId}/constraints
POST /courses/{courseId}/matchruns
GET /matchruns/{runId}/status
GET /courses/{courseId}/matchruns/{runId}/teams
GET /courses/{courseId}/matchruns/{runId}/metrics

Lambda never runs optimization directly. It only enqueues jobs.

6. Messaging Layer (AWS SQS)

SQS decouples user traffic from optimization workload.

Queue Design

Main Queue: MatchRunQueue
Dead Letter Queue: MatchRunDLQ

Message Attributes

runId
courseId
rosterSnapshotId
attemptCount
createdAt

Benefits

- Absorbs traffic spikes
- Enables retries
- Prevents API timeouts
- Ensures durability

7. Workflow Orchestration (AWS Step Functions)

Step Functions coordinates the lifecycle of each match run.

Workflow Stages

Stage 1 – Validate Context
Confirm roster snapshot and constraints exist.

Stage 2 – Fetch Inputs
Retrieve student profiles and configuration from DynamoDB.

Stage 3 – Compute Skill Weights (LLM)
Agent calls LLM to compute project-specific skill weights.

Stage 4 – Run Matching Engine
Agent performs greedy initialization and local improvement.

Stage 5 – Score + Explainability
Compute metrics and generate natural-language explanation.

Stage 6 – Persist Results
Write final teams and metrics to DynamoDB.

Stage 7 – Completion Update
Update run status and notify UI.

Failure Handling

- Automatic retries for transient failures
- DLQ routing for repeated failures
- Error reason stored in DynamoDB for UI display

8. Agent Microservice (ECS Fargate)

The Agent Microservice is containerized and deployed on ECS Fargate.

It performs all long-running and compute-intensive operations.

Responsibilities

- Watches for “Run Match” jobs
- Pulls roster and project data
- Calls LLM for weight computation
- Executes deterministic matching engine
- Computes metrics
- Generates explainability text

- Writes results to DynamoDB
- Updates run status

Internal Modules

InputLoader
WeightingEngine
MatchingEngine
ScoringEngine
ExplainabilityGenerator
RunLogger

Core Functions

computeSkillWeights(projectDescription, skillTaxonomy)
generateInitialTeamsGreedy(students, constraints, weights)
improveTeamsLocalSearch(teams, objectiveFn, maxIters)
scoreTeams(teams, students, weights)
generateExplainability(teams, metrics, weights)

Matching Logic

Hard Constraints:

- Every student assigned exactly once
- Fixed team size
- Full roster coverage
- No duplicates

Soft Objectives:

- Minimize skill variance
- Maximize schedule overlap
- Balance experience
- Spread leadership preference

Objective Function Example:

TeamScore = w1*SkillBalance + w2*ScheduleOverlap + w3*ExperienceDistribution + w4*LeadershipDistribution

Deterministic seed and consistent sorting ensure reproducibility.

9. Data Layer (DynamoDB)

DynamoDB is the system’s source of truth.

Tables

CoursesTable
StudentProfilesTable
RosterSnapshotsTable
MatchRunsTable
TeamsTable
RunLogsTable

Stored Data

- Course configuration
- Student skills and availability
- Match run metadata
- Final team assignments
- Scoring metrics
- Audit logs

Access patterns are optimized for instructor dashboards and run result retrieval.

10. Secrets and Configuration (AWS Secrets Manager)

LLM API keys and sensitive configuration values are stored in Secrets Manager.

Lambda and ECS access secrets via IAM roles.

11. CI/CD (GitHub Actions → AWS)

Frontend

Build → Upload to S3 → CloudFront invalidation

Lambda

Package → Deploy via AWS CLI or SAM

Agent

Build Docker image → Push to ECR → Update ECS task definition

Infrastructure can be defined using CloudFormation or Terraform.

12. End-to-End Flow

1. Student submits survey → Lambda validates → DynamoDB stores profile.
2. Instructor configures constraints → DynamoDB updated.
3. Instructor clicks Run Match → Lambda creates MatchRun + pushes SQS job.
4. Step Functions starts orchestration.
5. Agent retrieves data → computes weights → runs matching → scores → explains.
6. Results written to DynamoDB.
7. MatchRun marked COMPLETED.
8. UI polls status and displays teams + metrics + explanation.

13. Monitoring and Observability (CloudWatch)

- Lambda execution logs
- ECS container logs
- Step Function execution history
- SQS queue depth metrics
- Run duration statistics
- Failure rate monitoring

This ensures operational transparency and debugging capability.

14. Architectural Strengths

- Fully asynchronous heavy workflows
- Scalable serverless API
- Deterministic and explainable optimization
- Clear separation between API and agent
- Production-grade workflow orchestration
- Durable job handling
- Secure secret management
- Cloud-native scalability