

- Users → Web App
- Web App → Backend API
- Backend API → Profile Service → Database
- Backend API → Optimization Engine → Database
- Backend API → Scoring & Analytics → Database
- Backend Services → Logs/Monitoring

Builds candidate team configurations under constraints (team size, balance, overlap).
Scores candidates and selects the best overall team assignment.

Optimization Engine (AI Team Formation Agent)

- Builds candidate team configurations under constraints (team size, balance, overlap).
- Scores candidates and selects the best overall team assignment.

Profile Service

- Stores student survey responses (skills, experience, availability).
- Normalizes inputs into consistent scoring scales.

Backend API

- REST endpoints
- Validation & Routing

- A single REST API endpoint layer (label: "Backend API")
- Handles requests from the Web App
- Validates input and routes to services

AI Team Formation Agent (Optimization Engine)

Profile Service

- Student data

Scoring & Analytics

- Team explanations

Logs / Monitoring

- Capture API/service logs and optimization runs.
- Help debug issues and track performance (latency, failures, generation time).

Logs/Monitoring

PostgreSQL Database

- Students
- Availability
- Teams
- Scores

Database (PostgreSQL)

- Source of truth for students, courses, teams, and score breakdowns.
- Supports queries for dashboards, exports, and regeneration history.

TeamMatch System Architecture - Technology Breakdown

- Frontend - React / Next.js
- We are using **React with Next.js** to build the web application interface.

Why:

- Component-based architecture for scalable UI
- Fast rendering and clean state management
- Easy integration with REST APIs
- Production-ready build output

Purpose in our system:

- Students complete surveys
- Professors configure constraints
- Displays optimized team results and scoring transparency

API Layer - REST Backend (Node.js / Express)

We use a **Node.js backend with Express** to handle HTTP requests.

Why:

- Lightweight and fast
- Works well with JSON-based APIs
- Easy integration with PostgreSQL
- Simple routing and middleware support

Purpose:

- Receives frontend requests
- Validates and sanitizes input
- Routes requests to internal services
- Acts as the secure entry point to the system

Backend Services - Microservice Logic

Inside the backend, we logically separate responsibilities:

Profile Service

Handles:

- Storing survey data
- Normalizing skill values
- Managing availability inputs

Optimization Engine (Core AI Agent)

Implements:

- Constraint satisfaction logic
- Heuristic optimization
- Scoring function evaluation
- Selection of best team configuration

Technology:

- Implemented in Node.js (or Python if using optimization libraries)
- Uses combinatorial search or greedy heuristics

This is the "intelligence" of the system.

Scoring & Analytics Service

Responsible for:

- Calculating team balance metrics
- Generating score breakdowns
- Providing transparency explanations

This ensures our AI is explainable, not a black box.

Database - PostgreSQL

We use **PostgreSQL** as our relational database.

Why:

- Strong relational integrity
- Handles structured survey data well
- Supports joins for team aggregation queries
- Reliable and scalable

Stores:

- Students
- Courses
- Skill ratings
- Availability
- Team assignments
- Scoring breakdowns

Deployment / Cloud Infrastructure (if using AWS)

- Frontend hosted via S3 (static hosting)
- Backend containerized using Docker
- Deployed via AWS ECS
- Data stored in AWS RDS (PostgreSQL)
- Logs monitored via CloudWatch

Why:

- Scalable
- Production-grade infrastructure
- Isolated compute and data layers

How It All Works Together (Simple Flow)

1. Student submits survey → Frontend
2. Frontend sends request → Backend API
3. Profile Service stores data → PostgreSQL
4. Professor clicks "Generate Teams"
5. Optimization Engine reads from DB
6. Engine computes best configuration
7. Teams stored in DB
8. Results returned to frontend

<