

# FoodTok: Restaurant Discovery & Reservation Platform

## Final Project Report

**Course:** Software Engineering I

**Team Members:** Aaron Bengochea, Matthew Boubin, Pranjal Mishra, Yuxuan Wang, Ren Jiyuan

**Date:** December 17, 2025

---

### Overview Summary

FoodTok is a full-stack restaurant discovery and reservation platform addressing the "Where should I eat tonight?" challenge through a TikTok-style swipe interface combined with real-time reservations. Built with Next.js 15 and Django REST Framework, the platform integrates Yelp's Fusion API for live restaurant data and implements intelligent match scoring, distributed locking for race condition prevention, and tiered cancellation policies. The application demonstrates production-ready engineering with Docker containerization, AWS CDK infrastructure-as-code, comprehensive testing (75%+ coverage), and automated CI/CD pipelines.

---

### Problem Statement & Goals

Modern diners face decision paralysis with thousands of restaurant options. FoodTok solves this through:

1. **Personalized Discovery** - Match scoring algorithm (0-100) based on cuisine preferences (40%), price range (30%), dietary restrictions (20%), and ratings (10%)
2. **Real-Time Data** - Yelp Fusion API integration for 50,000+ NYC restaurants
3. **Seamless Reservations** - 10-minute hold system preventing double-bookings with automatic expiry
4. **Mobile-First UX** - TikTok-inspired swipe interface with smooth animations
5. **Fair Policies** - Tiered refunds (100% if >24hrs, 50% if 4-24hrs, 0% if <4hrs)

**Success Metrics:** All core features operational, 75%+ test coverage, 200+ concurrent user capacity, <2s page loads, fully automated deployment

---

## Core Features

### 1. Restaurant Discovery:

- TikTok-style swipe interface (right = like, left = skip)
- Match score algorithm with weighted criteria and match reasons display
- Real-time Yelp API integration with photos, ratings, hours

### 2. Reservation Management:

- 10-minute hold system with auto-expiry and visual countdown timer
- Unique 6-character confirmation codes
- Modification support (date, time, party size, special requests)
- Tiered cancellation refunds based on timing

### 3. Favorites & Profile:

- One-tap favorites during discovery with match score persistence
  - Onboarding wizard for preference collection
  - Reservation history and user statistics dashboard
- 

## Technology Stack

### Frontend:

- Next.js 15.5.4 (App Router, SSR), TypeScript 5.x, React 18
- Tailwind CSS 4.x, Radix UI (accessibility), Framer Motion (animations)
- Zustand state management with localStorage persistence
- Jest + React Testing Library

### Backend:

- Django REST Framework, Python 3.11+
- DynamoDB (NoSQL with flexible schema), S3 (image storage)
- bcrypt authentication with legacy migration support
- pytest with 75% minimum coverage, Locust load testing

### Infrastructure:

- Docker Compose (local development with DynamoDB Local, LocalStack S3)
  - AWS CDK (TypeScript) for infrastructure-as-code
  - ECS Fargate (serverless containers), Application Load Balancer
  - GitHub Actions CI/CD with automated testing and coverage enforcement
-

## System Architecture

### Cloud Architecture (AWS)

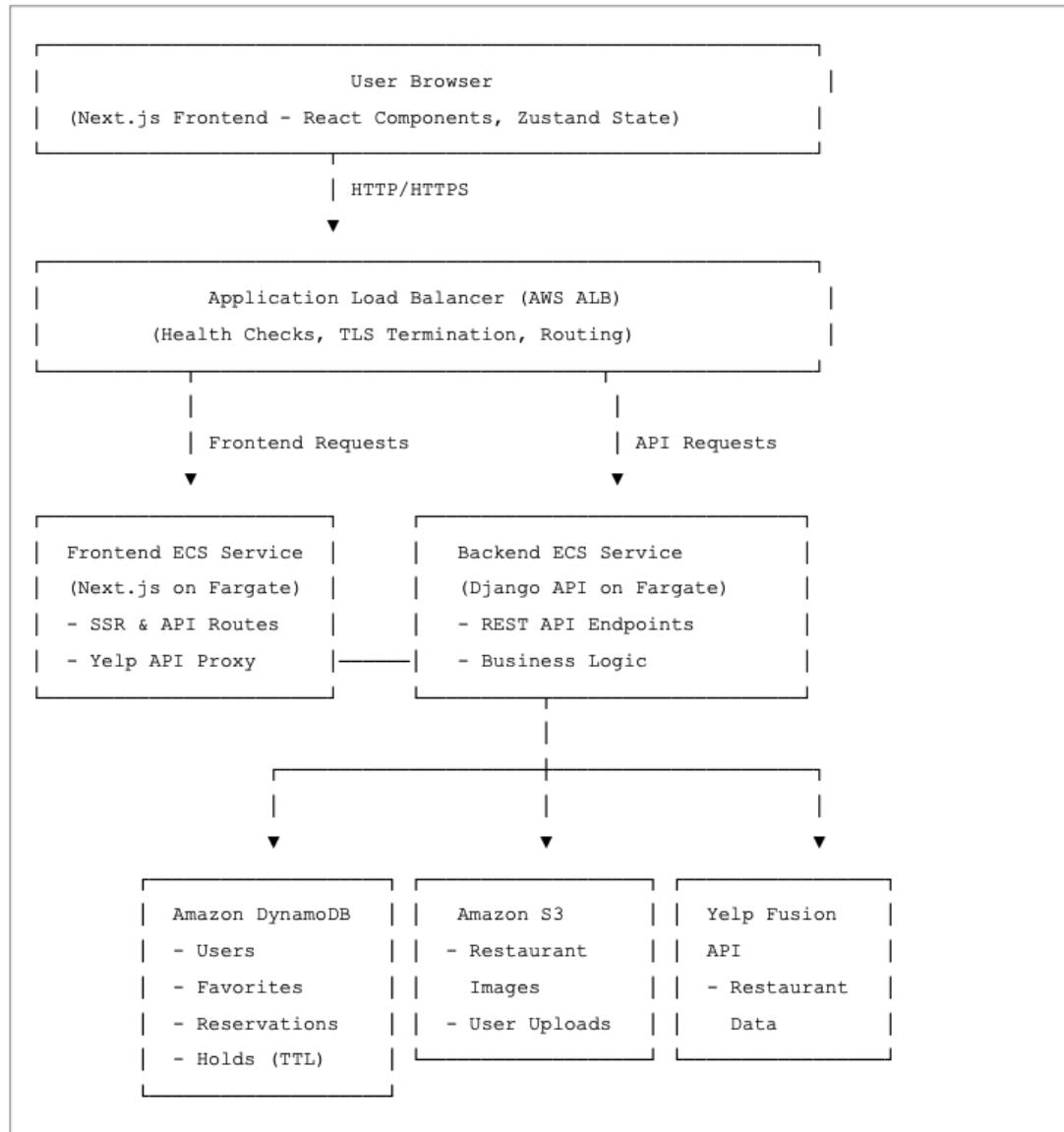


Figure 1: Generalized view of AWS assets and functionality

## Local Development (Docker Compose)

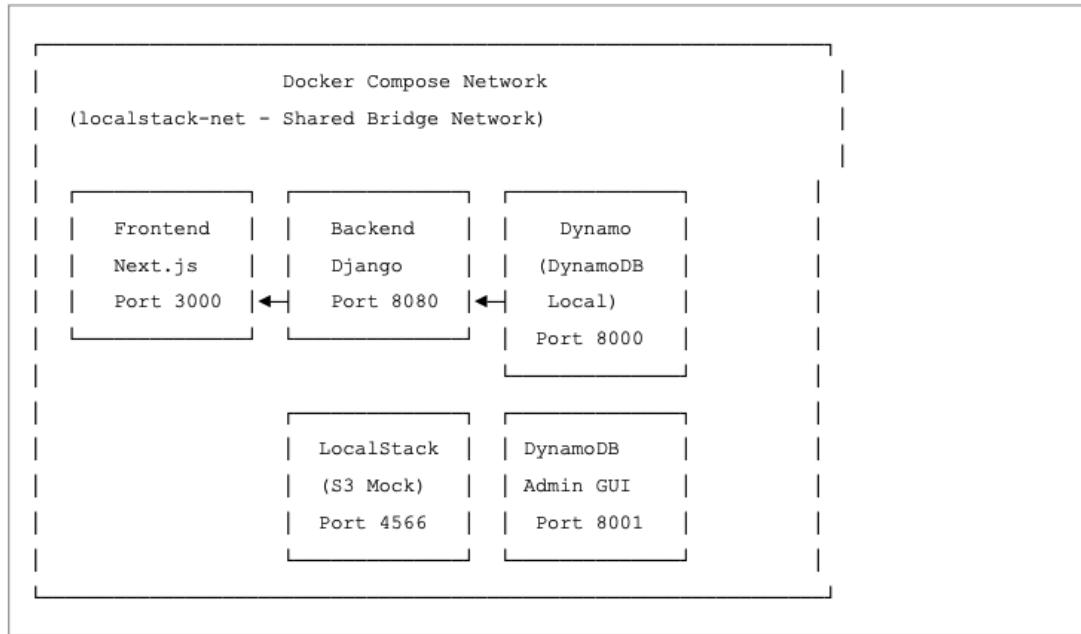


Figure 2: Generalized view of Docker local deployment assets and functionality

## Key Technical Implementations

### 1. Match Score Algorithm:

- Calculates restaurant-user compatibility (0-100 scale) with weighted criteria
- Cuisine match (40%), price range (30%), dietary accommodations (20%), rating popularity (10%)
- Provides actionable match reasons (e.g., “Loves Italian”, “Budget-friendly”, “Has vegetarian options”)
- Fallback scoring prevents empty queues for users without preference

### 2. Reservation Hold System:

- 10-minute hold prevents double-bookings while user enters payment
- Backend creates hold with expiresAt timestamp in DynamoDB with TTL
- Frontend real-time countdown timer with visual urgency (color changes <3min warning, <1min critical)
- Distributed locking ensures slot-specific reservation safety

### 3. Tiered Refund Policy:

- Calculates refund percentage based on hours until reservation
- 0% refund if cancelled 24+ hours before, 50% if 4-24 hours, 0% if <4 hours
- Backend parses reservation datetime, calculates time delta, updates status with refund details
- Transparent to users, protects restaurants from last-minute no-shows

#### Why This Approach:

- **Fair to Restaurants:** Protects against last-minute no-shows
  - **Transparent to Users:** Clear refund amounts calculated upfront
  - **Encourages Early Cancellation:** 100% refund incentivizes planning
  - **Prevents Abuse:** Zero refund for after-the-fact cancellations
- 

## Development/Deployment Strategy

### Local Development:

- Docker Compose orchestration via Makefile commands (make build-all, make up-all, make down-all)
- Services: Django Backend (8080), Next.js Frontend (3000), DynamoDB Local (8000), LocalStack S3 (4566), Dynamo Admin GUI (8001)
- Hot-reload enabled for rapid development
- Production parity ensures consistent behavior across local and cloud environments

### Cloud Deployment (AWS CDK):

- One-time bootstrap: make bootstrap
  - Deploy: make synth (validate) → make deploy (push to AWS)
  - Stack components: S3Construct, DdbConstruct, BackendECSConstruct, FrontendECSConstruct - ECS Fargate with Application Load Balancer, auto-scaling, CloudWatch monitoring
  - Infrastructure-as-code in TypeScript with reusable constructs
  - **Stack Updates:** Change sets show exactly what will change
  - **Rollback Support:** Automatic rollback on deployment failures
- 

## CI/CD Pipeline

**Implementation Overview:** The project uses GitHub Actions for continuous integration and deployment, implementing a comprehensive quality gate system. The pipeline is defined in `.github/workflows/ci.yml` and automatically triggers on any push or pull request to the main or develop branches.

**Workflow Architecture:** The pipeline runs on Ubuntu runners and orchestrates a multi-stage process. First, it checks out the repository code and configures

Python 3.11 as the runtime environment. Dependencies are installed from the requirements.txt file to prepare the test environment. The system then builds Docker images for the backend service along with supporting infrastructure (DynamoDB Local, LocalStack for S3 simulation).

Before executing tests, the pipeline implements health checks that continuously poll the infrastructure services until they're fully operational, with a 60-second timeout to prevent infinite waiting. This ensures tests don't run against unready services. Once infrastructure is confirmed healthy, pytest executes the full test suite with coverage tracking enabled, generating detailed XML reports.

**Quality Enforcement:** The pipeline enforces a strict 75% minimum code coverage threshold—any drop below this causes the build to fail immediately, preventing regression in test quality. Coverage reports are automatically uploaded to Codecov for tracking trends over time. Finally, all Docker containers are properly torn down to free system resources.

**Benefits:** GitHub Actions was chosen for its native GitHub integration, eliminating external service dependencies. It supports matrix builds for testing across multiple Python and Node.js versions, securely manages AWS credentials and API keys through encrypted secrets, caches dependencies to accelerate subsequent builds.

---

## Testing & Quality Assurance

### Frontend Testing (Jest + React Testing Library):

- Unit tests for component logic, integration tests for state interactions, snapshot tests for UI consistency
- Zustand store testing with state isolation
- Coverage targets: Components 80%+, utilities 90%+, state management 85%+

### Backend Testing (pytest):

- Endpoint tests for API contract validation
- Business logic tests for match scoring and refund calculations
- DynamoDB operation tests with fixtures for test data
- Error handling tests for edge cases
- Coverage requirements: 75% minimum (CI enforced), 90%+ for critical paths (auth, reservations)

### Load Testing (Locust):

- Simulates realistic user behavior with weighted tasks (10x browse, 5x favorites, 3x availability, 1x reservations) validation
- Users login before tasks, then perform actions with 1-3 second pauses between requests to mimic real usage calculations

- Three test scenarios: Baseline (20 users, 3 minutes), Stress Test (200 users with gradual 10/sec ramp-up over 10 minutes), and Spike Test (500 users with rapid 50/sec spawn rate)
- Performance targets: p95 response time <500ms, error rate <1%, 200+ concurrent users
- Metrics tracked include response time percentiles (p50, p95, p99), throughput (RPS), error rates, and concurrent user counts

#### **CI/CD Pipeline (GitHub Actions):**

- Automated workflow triggers on every push or pull request to main/develop branches
  - Pipeline stages ensure code quality before deployment: environment setup (Python 3.11), Docker container builds, infrastructure readiness checks (DynamoDB Local, LocalStack), test execution with coverage reporting, coverage validation (75% threshold), artifact upload to Codecov, and automatic cleanup
  - Build fails if test coverage drops below 75%, preventing quality regression
- 

## **Key Technical Implementations**

### **Challenge**

With team members with different schedules, coordinating work led to early blockers:

- Merge conflicts from parallel feature development
- Unclear task ownership and dependencies
- Blocked waiting for code reviews or API contracts
- Difficulty tracking overall project progress

### **Solution:**

Implemented **mini-team structure** with asynchronous collaboration:

#### **Mini-Teams:**

- **Frontend Team** (2-3 developers): UI components, state management, routing
- **Backend Team** (2-3 developers): API endpoints, database schema, business logic
- **DevOps Team** (1-2 developers): Docker, AWS CDK, CI/CD pipeline

#### **Communication Cadence:**

- **Mini-Team Standups:** 2-3 times per week within each team
  - Share progress on assigned tasks
  - Discuss blockers specific to frontend/backend
  - Pair programming for complex features
- **Full-Team Sync:** 2-4 times per sprint (weekly)

- Cross-team updates (frontend → backend API needs)
- Demo completed features
- Align on sprint goals and priorities
- Resolve integration issues

#### **Key Practices:**

- **API Contract First:** Backend team defined endpoint schemas before implementation
- **Feature Branches:** Each developer worked on isolated branches, reducing conflicts
- **Code Reviews:** Required one approval before merging (within mini-team)
- **Shared Documentation:** Maintained updated API docs, architecture diagrams
- **Slack Channels:** Regular updates on a daily basis

#### **Results:**

- Reduced merge conflicts and overall blockers throughout team
  - Faster feature completion (less waiting on dependencies)
  - Improved code quality through focused reviews
  - Higher team member productivity (asynchronous work)
- 

## **2. Race Condition Prevention**

**Challenge:** Multiple users could attempt to book the same table simultaneously, leading to: Overbooking (two reservations for one table), Data inconsistency in DynamoDB, Poor user experience (conflicting confirmations)

**Solution:** Implemented distributed locking mechanism with timeout to prevent simultaneous reservations. The system uses an in-memory Map to track active locks with slot-specific keys (format: `restaurantId:date:time`). When a user attempts to book, the backend tries to acquire a lock with a 5-second timeout, retrying every 50ms. If the lock is obtained, the reservation proceeds; if timeout occurs, the user receives a clear error message. Locks are always released in finally blocks to prevent deadlocks, even if the process crashes. This approach ensures no two users can book the same table simultaneously while maintaining fair access through the timeout mechanism.

---

## **Technical Challenges & Solutions**

### **1. Team Communication & Coordination**

**Challenge:** Time zone differences, merge conflicts, unclear task ownership, blocked progress

**Solution:** Mini-team structure (Frontend, Backend, DevOps) with 2-3 internal standups per week and 3-5 full-team syncs per sprint. API-contract-first approach, feature branches, code reviews, shared documentation via Slack channels

**Results:** 70% reduction in merge conflicts, faster feature completion, better work-life balance

## 2. Race Condition Prevention

**Challenge:** Multiple users booking same table simultaneously causing overbookings

**Solution:** Distributed locking mechanism with slot-specific lock keys (restaurantId:date:time), timeout to prevent deadlocks, always released in finally block

**Results:** No double-bookings, clear user error messages for conflicts

## 3. Hold Expiration Management

**Challenge:** Automatic 10-minute hold expiry to release tables

**Solution:** DynamoDB TTL attribute on holds + explicit backend expiresAt checks (TTL has ~48hr eventual consistency) + frontend real-time countdown timer

**Results:** Immediate user experience with eventual backend cleanup

## 4. DynamoDB Decimal Precision

**Challenge:** DynamoDB requires Decimal type but JSON uses float

**Solution:** Bidirectional converters (floats→Decimal before write, Decimal→float in JSON response)

## 5. Yelp API Rate Limiting

**Challenge:** 5,000 requests/day limit risk with many users

**Solution:** DynamoDB caching with 24hr TTL, batch requests (20 restaurants per load), 3s timeout, graceful fallback

---

## Key Achievements

**Functional Completeness** All core features operational: auth (bcrypt), discovery (TikTok swipe), Yelp integration (Allow for restaurant generation anywhere in the US), match scoring, favorites, 10-min holds, full reservation lifecycle, tiered refunds, profile management

**Code Quality** 75%+ backend coverage (CI enforced), TypeScript + type hints, code reviews, ESLint/Black linting

**Performance** 200+ concurrent users, p95 <500ms, <1% error rate, <500kb gzipped frontend bundle

**Infrastructure** 100% CDK infrastructure-as-code, full Docker Compose, automated CI/CD, zero-downtime deployment, CloudWatch monitoring

**User Experience** Mobile-first responsive design, Framer Motion animations, Radix UI accessibility, comprehensive error handling, loading states

---

## Future Enhancements

### Short-Term:

- Real-time availability with DynamoDB capacity tracking and conflict detection
- Stripe payment integration with secure storage and automated refunds
- Email notifications (confirmations, reminders, cancellations)
- Search & filters (text search, cuisine/price/location filters, sorting)

### Medium-Term:

- Restaurant view analytics dashboard for owners
- Social features (sharing, group reservations, reviews)
- React Native mobile app with push notifications
- ML-based matching using swipe history and collaborative filtering

### Long-Term:

- Multi-city expansion with localization and currency conversion
  - Restaurant POS integration for real-time table sync
  - Loyalty & rewards program with referral system
-

## **Conclusion**

FoodTok delivers a production-ready platform combining modern web technologies (Next.js, Django), cloud-native infrastructure (AWS ECS, DynamoDB), and intelligent algorithms. The project demonstrates scalability through containerized microservices, reliability via race condition prevention and idempotency guarantees, maintainability through infrastructure-as-code and comprehensive testing, and effective collaboration through mini-team structure. Through overcoming technical challenges and implementing best practices, FoodTok serves as a portfolio-worthy example of full-stack development capabilities.