Eventak Production Readiness Audit

Mock Data & Fallback Analysis Report

Repository: tasipred/eventak.
Audit Date: February 2025
Status: NOT PRODUCTION READY

1. Executive Summary

This audit was conducted to identify mock data, fallback mechanisms, and non-production code in the Eventak system. The analysis revealed multiple critical issues that prevent the system from being considered production-grade. The codebase contains extensive mock services, hardcoded test credentials, simulation logic, and fallback mechanisms that must be addressed before production deployment.

Category	Count	Severity
Mock Service Files	3	CRITICAL
Hardcoded Test Credentials	3 users	CRITICAL
Simulation/Fallback Logic	5 locations	HIGH
Mock Data Generators	2 files	HIGH
Quick Test Access UI	1 location	MEDIUM

Table 1. Summary of Findings

2. Critical Issues (Must Fix Immediately)

2.1 Mock Firestore Database Service

File: `server/services/mockFirestore.js`

This file implements a complete in-memory mock database that mimics Firestore behavior. It includes mock implementations for collections, documents, queries, and batch operations. The mock stores all data in memory and loses everything on restart. While the system has a flag to switch between mock and real database, the presence of this file and its usage patterns indicate testing code that should not exist in production.

Key Problems:

- In-memory storage with no persistence

- Mock admin SDK with fake FieldValue operations

- Triggered by environment variable USE_MOCK_DB=true

2.2 Frontend Mock Backend Service

File: `src/services/mockBackend.js`

This is the most critical issue. The frontend uses a completely mock backend that stores data in localStorage. It includes hardcoded test users, mock vendor generation, auto-bidding simulation, and fake API responses. The AuthContext.jsx directly imports and uses this mock service, meaning the frontend is NOT connected to the real backend API.

Hardcoded Test Users:

Email	Password	Role
admin@test.com	123	admin
client@test.com	123	client
vendor@test.com	123	vendor

Table 2. Hardcoded Test Credentials in mockBackend.js

2.3 Mock Vendor Generator

File: `src/services/mockVendors.js`

This file generates fake vendor data programmatically. It creates 100+ mock vendors with random Arabic names, fake phone numbers, random service categories, and placeholder portfolio images from Unsplash. These vendors are used by the mockBackend to simulate vendor matching and bidding.

Problems:

- Generates 3 vendors per service category per city (100+ total)

- Uses external Unsplash images without permission

- Creates fake phone numbers with random digits

3. High Priority Issues

3.1 AI Service Fallback Mechanisms

Files: `gateway/services/gemini.py`, `gateway/services/deepseek.py`

Both AI services (Gemini and DeepSeek) have fallback methods that return mock responses when the API fails or is not configured. These fallbacks return generic Arabic error messages but still provide structured data that could be processed incorrectly.

Example from deepseek.py (lines 294-302):

```
def _mock_fallback(self, text: str, debug_error=''):
    return {
        'intent': 'NEW_REQUEST',
        'service_category': 'Unknown',
        'location': 'Unknown',
        'date': 'Unknown',
        'missing_info': ['service_category'],
        'reply_message': '...'
    }
```

3.2 Backend Service Request Fallback

File: `gateway/services/backend.py` (lines 40-42)

When the Node.js backend is unreachable, the gateway service returns a random request ID instead of properly failing. This could lead to customers tracking non-existent requests.

```
except Exception as e:
    print(f'Failed to create request: {e}')
    import random
    return {'requestId': str(random.randint(100000, 999999))}
```

3.3 Hardcoded Default Client Phone

File: `gateway/services/store.py` (line 10)

The StoreService class has a hardcoded fallback client phone number that serves as a default. This appears to be a test phone number that should not exist in production.

```
self.last_active_client = '966551315886' # Default Fallback
```

3.4 Random Bid Amount Calculation

File: `server/controllers/requestsController.js` (lines 4-8)

The bid amount calculation uses Math.random() to generate prices. This is simulation logic that should not exist in a production system where vendors should set their own prices.

```
const calculateBidAmount = (budget) => {
    const base = budget || 5000;
    return Math.floor(base * (0.8 + Math.random() * 0.4));
};
```

4. Medium Priority Issues

4.1 Quick Test Access Buttons in Login UI

File: `src/pages/Login.jsx` (lines 82-90)

The login page contains 'Quick Access for Tester' buttons that auto-fill test credentials. While convenient for development, these should be removed or conditionally rendered only in development mode.

```
{/* Quick Access for Tester */}
<div className='mt-8 pt-6 border-t border-dashed border-gray-200'>
    <p className='text-xs text-center text-gray-400 mb-3'>[redacted]</p>
    <div className='flex gap-2 justify-center text-xs'>
        <button onClick={() => { setFormData({ email: 'client@test.com',
password: '123' }); }}>...
    </div>
</div>
```

5. Recommendations

5.1 Immediate Actions Required

1. Replace mockBackend.js with real API calls: The frontend AuthContext and all components must be updated to call the real backend API endpoints instead of using localStorage mock data.

2. Remove mockFirestore.js: Ensure the production environment never has USE_MOCK_DB=true and consider removing the mock file entirely from the codebase.

3. Delete mockVendors.js: All vendor data must come from the real database. Remove this generator completely.

4. Remove hardcoded credentials: Delete the test users from mockBackend.js and ensure the database has proper user authentication.

5.2 Architecture Changes Needed

1. Implement proper API service layer: Create a dedicated API service (e.g., api.js) that handles all HTTP communication with the backend, with proper error handling and authentication.

2. Remove simulation logic: The random bid generation, auto-bidding simulation, and fake vendor matching must be replaced with real business logic.

3. Environment-based configuration: Use proper environment variables to distinguish between development, staging, and production environments.

5.3 Fallback Strategy

For AI services, instead of returning mock data on failure, implement proper error handling that informs the user of the issue and suggests retry. For backend connectivity, implement circuit breaker patterns and proper health checks rather than returning fake request IDs.

6. Complete List of Affected Files

File Path	Issue Type	Action
server/services/mockFirestore.js	Mock Service	DELETE
src/services/mockBackend.js	Mock Service	DELETE
src/services/mockVendors.js	Mock Generator	DELETE
src/context/AuthContext.jsx	Uses Mock	REFACTOR
src/pages/Login.jsx	Test UI	CLEANUP
gateway/services/backend.py	Fallback	FIX
gateway/services/gemini.py	Fallback	FIX
gateway/services/deepseek.py	Fallback	FIX
gateway/services/store.py	Hardcoded	FIX
server/controllers/requestsController.js	Simulation	REFACTOR
server/app.js	Mock Seeding	CLEANUP

Table 3. Complete List of Affected Files and Required Actions

7. Conclusion

The Eventak system is NOT PRODUCTION READY. The codebase contains extensive mock data, hardcoded test credentials, simulation logic, and fallback mechanisms that must be addressed before any production deployment. The most critical issue is that the frontend is currently using a mock backend (localStorage) instead of connecting to the real Node.js API.

Priority should be given to removing the mock services (mockBackend.js, mockFirestore.js, mockVendors.js) and refactoring the frontend to use proper API calls. All hardcoded credentials must be removed, and proper error handling should replace the current fallback mechanisms.